Aalto University

School of Science

Master's Programme in Computer, Communication and Information Sciences

Workneh Tefera

Evaluation of Progressive Web Application to develop an Offline-First Task Management App

Master's Thesis
Espoo, September 18, 2019

Supervisor: Professor Petri Vuorimaa
Advisor: Jaakkola Noora

ABSTRACT

Author	Workneh Tefera Tamire
Title	Evaluation of Progressive Web Application to develop an Offline-First Task Management App
Number of Pages	vii + 68
Date	September 18, 2019
Degree	Master of Science (Technology)
Degree Programme	Master's Programme in Computer, Communication and Information Sciences
Major	Computer Science
Specializations	Web Technologies, Applications and Science
Supervisor	Professor Petri Vuorimaa
Advisor	Jaakkola Noora

Adding more features to web apps progressively to feel and work like native apps is a recent design philosophy. This research study was conducted on Progressive Web Apps (PWAs) to develop an offline-first task management app. The main idea of PWAs is enhancing web apps gradually by adding new features to existing or new web applications based on browser support. Hence, the main goal of this study was to explore PWAs features available in the browsers, which could enable developers to implement an offline-first web applications.

The study was carried out in two ways. The first part thoroughly reviewed theories about the web, native apps, and PWAs features that are available in browsers. An offline-first task management app called Annual Clock was designed and implemented for Aalto university in the second part of the thesis. To implement the app, React.js was used for the frontend and Node.js for the backend. The application was fully implemented using most PWA features to work offline, install on home screen, load fast, send push notification, etc.

To conclude, the implemented app presents solutions to most offline-first web app problems and will be used as a valuable reference for the university and other researchers who would like to develop an offline-first PWAs. Nonetheless, there are many areas left untouched and need to be solved in the future like saving images, audios and videos in browser storage, increasing the size of browser storages for large data and background synchronization support by all browsers.

Keywords	App Manifest, App Shell, Background synchronization, Offline-first, Progressive Web App, Push Notification, React.js, Single-page application, Service Worker, Web application
Language	English

Acknowledgments

In the first place, I would like to thank Almighty God for giving me the opportunity, ability, strength, and knowledge to carry out this research study and to proceed and accomplish it adequately. Without his miracles, this research study would have been difficult.

Following that, I would like to express my heartfelt gratefulness to my supervisor professor Petri Vuorimaa, who backed me during this study. He recommended me to work on this fascinating topic with constant assistance and encouragement. With his enormous experience and enthusiasm in the field of Progressive Web Apps, he assisted me solve all issues which came up.

Further, I would like to thank my advisor Jaakkola Noora for her guidance and assistance during the implementation phase of the project. I would like to thank her for promptly answering all my questions about the Annual Clock.

My thanks also go to my friends for proofreading the thesis, for all the suggestions and excellent peer support.

Finally, I would like to thank my families for their love, encouragement, and support.

Abbreviations and Acronyms

API	Application Programming Interface
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CLI	Command Line Interface
CRUD	Create, Read, Update and Delete
CSS	Cascading Style Sheets
DOM	Document Object Model
ECMA	European Computer Manufacturer's Association
GPS	Global Positioning System
JSX	JavaScript Syntax Extension
HTML	HyperText Markup Language
HTTP	HyperText Transport Protocol
HTTPS	HyperText Transfer Protocol Secure
ID	Identifier
IndexedDB	Indexed Database API
JSON	JavaScript Object Notation
MVC	Model View Controller
NPM	Node Package Manager
SDK	Software Development Kit
PWAs	Progressive Web Applications
SEO	Search Engine Optimization
SPA	Single-Page Application
SSL	Secure Sockets Layer
UI	User Interface
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WHATWG	Web Hypertext Application Technology Working Group
WICG	Web Platform Incubator Community Group

Contents

1. Introduction

In recent years, the way companies design products and services are directly or indirectly affected by technological advancement. The adaptation of the internet as a business platform by most companies forced the need to develop fast, engaging, and reliable applications. As a result, exceptional innovation, particularly, the introduction of new technologies in the mobile web has enabled features that were impossible before into being [1]. Hence, some features that are previously only available in native apps are also possible in web apps.

According to statistics, in the past few years, there has been increase in the consumption of mobile applications in terms of reliability, growth, and engagement as compared to web applications [2]. Hence, application developers and business firms most of the times prefer to build native mobile applications. Nonetheless, developing, distributing and maintaining native apps require more time and resources. Furthermore, the user's reachability and publishing native app is a tiresome process. App users need to obey many steps to use native apps like signing up to the individual app store, verifying the available space in their device, internet access to download the app, etc, which is in most case time consuming and discouraging [3]. As a result, most companies prefer cross-platform developers instead of hiring for each platform [4].

As stated above, both technologies have their own shortcomings and hence, there is a need to unify the web and native apps by taking the best out of them. Therefore, the unifying technology that enabled this to happen is the Progressive Web Apps (PWAs), a term, which is very hot these days [5]. PWA is a set of features of technologies that can be added to existing web applications based on a set of concepts and requirements to be native applications. PWA was introduced by Google in May 2016 in San Francisco to address many of the features that web app does not have.

PWA is a term referring to a couple of features added to any web to progressively enhance the look and feel like native mobile apps. Background synchronization, offline capability, accessing the camera, an icon in the home screen, push notifications, accessing locations, which are previously only available in native mobile apps are possible in PWAs [6]. All these features are possible in PWAs because of the use of App shell, Service Worker, Web App Manifest and

Push Notifications. As a web app, PWA can be employed in most platforms increasing reachability, decreasing the human resource and budget of developing web apps [7].

1.1 Motivation

The aim of this research study is to analyze and assess PWAs features and compare its performance over the web and native apps. Further, a Progressive Task Management App called Annual Clock will be developed for Aalto University and its performance will be compared with the existing web and native apps. The application will be fully implemented using most PWA features.

Furthermore, the thesis tries to analyze PWAs, evaluate the technology and limitations and implement the application. In addition, the study will try to illustrate best practices and prospects of PWA and prevailing challenges it is facing. Moreover, the app is planned to improve user experience by minimizing the social and economic costs of Web and native apps. Finally, the study will make some suggestions on existing technologies and recommend further research areas.

1.2 Research Goals

Based on the motivations stated in the previous sections of the study, the objective of this study is to find out an answer to:

- What browsers features are available to develop a PWA?
- How to minimize the social and economic cost of web app by developing a PWA?
- How to implement the Task Management App in compliance with the requirements set by the PWAs?

The first two research questions will be answered in the theoretical part of the thesis whereas the last question will be answered in a technical part of the study. To answer the third question, a Task Management App called annual clock is developed and most PWA features are implemented in the app. The app is fully functional and different auditing metrics are used to measure user experience.

1.3 Thesis structure

This thesis is arranged as follows: Chapter 2 is a literature review about the web, native apps, and PWAs. Chapter 3 evaluates core building blocks of PWA features that are available in browsers. Literature review about Service Workers, App Shell, Web App Manifest, Push Notifications, IndexedDB, Background Synchronization, and Geo-locations will be analyzed in this section. Requirement analysis to implement the application will be compiled in Chapter 4. Chapter 5 is about the implementation of the app. Chapter 6 evaluates the implemented app. The pros and cons of developing a PWA will be discussed in chapter 7. And finally, Chapter 8 draws a conclusion of this study and point out a recommendation for future work.

2. Literature Review

PWAs is a new concept and academic contribution to this field is very confined. There are a limited number of books and theses works in this technology. Search results regarding PWAs in Google Scholar returned a very limited number of results. Hence, this chapter tries to review different sources concerning this technology. However, before deeply heading first into PWAs, some key difference between native apps and web apps will be discussed in this section of the study.

2.1 Comparison of Native Apps and Web Apps

Applications could be categorized as being either native or web. Nonetheless, there are also other applications that lie between the two and these applications are often called hybrid applications, which utilizes the mixture of the two applications [8]. As its name illustrates, web applications can be developed using pure web code such as HyperText Markup Language (HTML), Cascading Stylesheet (CSS), and JavaScript and utilize the devices web browser to display the output [8]. Using web applications as compared to native applications have many core benefits. One of this benefit is that the user does not need to search for the application in the app stores and download it in their device.

On the other hand, native applications make use of the native Software Development Kit (SDK) of their specific platform and the native language of the device [9]. According to Tal Ater, to utilize native applications, users need first to search the app in the websites using online ads and then visit the app store to install the application in their respective device [10]. Hence, the processes of installing native apps involve many steps and require the users to give consent to the app and wait until the app finishes download [10]. In case the available memory in the user's device is less than what the app requires, the user needs either to delete other existing apps to free space or leaves the app uninstalled.

As described above, the hurdle of installing a native app is very high involving many steps. Different companies use various marketing strategies to convince mobile users to install their apps. For instance, LinkedIn requests mobile website users to download native apps using full-screen modal [10]. As a result, most users reject to download native apps instead prefer to use

a mobile website. Hence, due to low cross-platform support, native apps are less favorable than web apps.

Another downside of native applications is that native apps do not update regularly [11]. Hence, unlike web applications, which can be refreshed by reloading the web page, updating native applications require the direct involvement of the user either to update manually or activate automatic update from the app store. Furthermore, it is easy to find web applications using web search engines and add into the home screen whereas finding native apps directly from the website is challenging and require providing a link to app store [11].

Furthermore, native applications are platform specific, which results in mobile platform fragmentation problem. The main reason for the fragmentation is that codes written for Android app cannot be used for an Apple iOS app since Java is used to write the code in Android whereas Objective-C is used in iOS app [12]. As a result, due to fragmentation, native apps development and maintenance for multiple platforms is very challenging since it requires high development, maintenance and testing times [13].

Nevertheless, as compared to Web applications, native apps have full access to the device's hardware sensors. Moreover, these apps have better performance, greater functionality and user experience than web apps [11].

To summarize, native applications are coded in the devices native language and run directly on the device, whereas web applications are coded in HTML, CSS, and JavaScript and served through the internet and run through the browser. Hybrid applications like web applications are coded in HTML, CSS, and JavaScript but run via an invisible browser packaged into the native applications and can be distributed for any supported platform [14]. Furthermore, native apps access the native Application Program Interfaces (APIs) and distribute through app stores, whereas web apps run on different platforms and hybrid applications make use of the better of the two applications [11].

2.2 Progressive Web App

PWAs is a term that describes a set of new technologies, which enable to develop an application that runs in web browsers. According to Tal Ater, most elements of PWA technologies were previously known only from native mobile applications [10]. It is generally a term referring to a couple of features added to any web application to progressively enhance to feel and work like a native mobile app. Progressive enhancements like background synchronization, offline capability, push notifications support, accessing the device camera, an icon in the home screen, accessing locations and security, which were previously only available in native mobile apps were possible with PWAs [4]. Hence, the term PWAs means to progressively enhance web apps to look and feel like native apps.

Before four years, the share of native apps in the market was massive. According to the 2015 U.S. mobile app report, native apps were a powerful force in the daily life of society [15]. Figure 2.1 below shows the share of native apps and mobile web in 2015 in the USA.

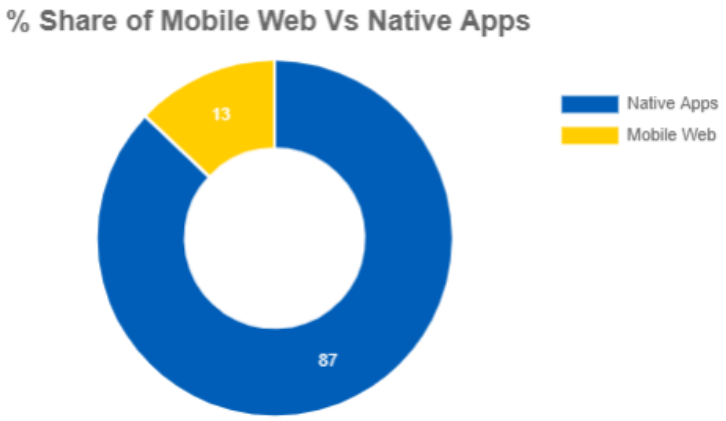

Figure 2.1: Mobile Web Vs Native Apps [15]

However, though the share of native apps is huge as illustrated in the figure (87%), people spend most of their times on the mobile web [15]. The question is why people spend more on the mobile web while native apps are better? The main reasons that force mobile web consumers to spend their times are Push Notification brings them back to the web, home screen icons make access to the web easy, accessing devices features like camera and possibility of offline functionality.

Furthermore, from the developers' perspective, building a native app is more difficult than the mobile web. The main reason for this is that developers need to learn two different programming languages to develop a native app, one for iOS and another for Android [12]. However, with Progressive Web, it is easy to develop a mobile web with the language the programmer already knows. In addition, although the above statistics states that the share of native apps is 87%, most users use the top three apps in their device: Google, Facebook, and Twitter [15]. Hence, the time users spent in other native apps is minimal and the average user does not install new apps unless the user buys a new device. Figure 2.2 below depicts the reach of native apps and mobile web in the U.S in 2015.

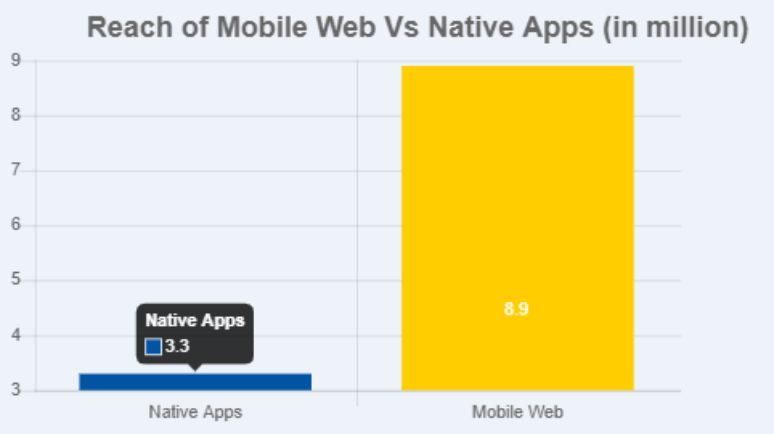

Figure 2. 2: Reach of Apps Vs Mobile Web in million [17]

As illustrated in the above bar chart, mobile web has 8.9 million reaches while native apps have only 3.3 million reaches [17]. Hence, developing a PWA helps to benefit the best of native apps and a traditional web page. Figure 2.3 below summarizes the main difference between native apps and the traditional Web.

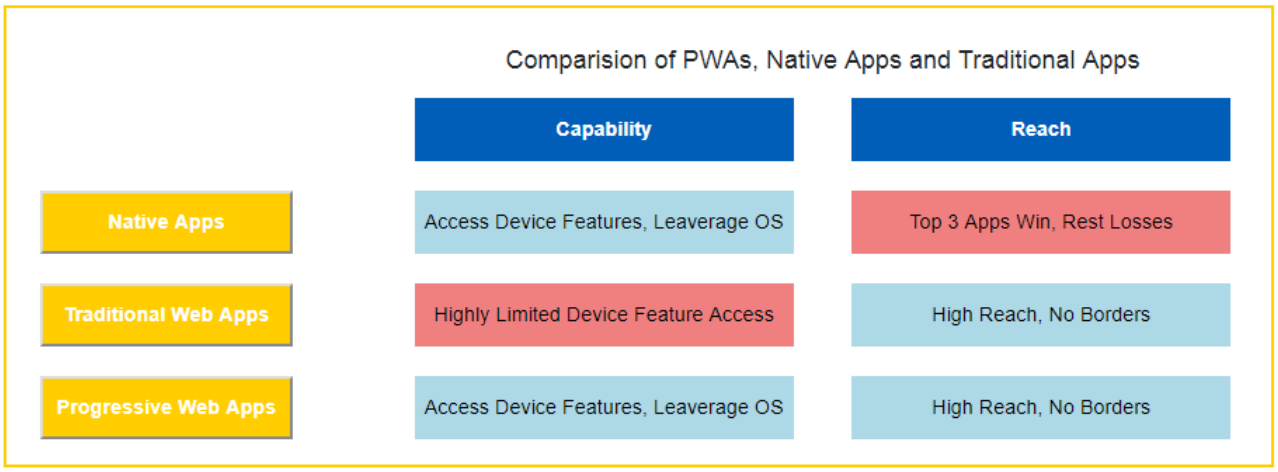

Figure 2.3: Comparison of PWAs, Native Apps and traditional Web pages

As shown in the figure above, PWAs combine the best of Native Apps and Traditional Web Pages. According to the Google developers' website, there are three main things, which summarize progressive web apps:

- **Be reliable**: which means load fast and provide at least some portion of the application to work offline [16].
- **Fast**: respond quickly to user's action not only on initial app load, but also to user actions. Further, it means to provide animations and native mobile app features [16].
- **Engaging**: on most mobile devices, PWAs performs like a native app. Offer features like push notification even if the application is closed [16].

The above three features of PWA diminishes the distinctions between a web application and a native application. Furthermore, according to Google documentation for PWAs, the following are the main properties of PWAs [16]:

> *"An application should be responding quickly to user interactions with silky smooth animations and no janky scrolling. It should load instantly and show a downasaur, even in uncertain network conditions." An application can be launched from the user's home screen; service workers enable a PWA to load instantly, regardless of the network state (offline modus). "An application should feel like a natural app on the device, with immersive user experience."*

Which implies the user's home screen can be used to install a PWA and it can be opened in full screen mode [16].

Tal Ater discussed the above main properties of PWA in his books by breaking down further based on their features as follows [10]:

1. **Offline First**

   According to Tal Ater, availability of the application regardless of being dependent on the network connection is the main feature of PWA [10]. PWA works like native apps once the user browses the application or installs the link in the user's home screen. However, in case of native apps, the user needs to install the application in the respective device, unlike PWAs, which work by clicking the shortcut icon from the user's home screen, or by inputting the URL of the web app in the web browser. The main thing that enabled this to happen in PWA is a Service Worker that is registered automatically and manages and downloads all static assets when the user enters a PWA [10].

2. **Load Fast**

   Using Service Worker enables static resources of the application to download once when the user visits the website. This is the only time, which requires an internet connection to load static assets and following that the application starts to be used by the user even without an internet connection [10]. As a result, since most resources are already in the client's device, visiting the web application next time does not take long [10].

3. **Push Notifications**

   It is one of the features of native applications before. However, it is possible to send push notification using PWA even after the user closed the browser tab and visited the application [10].

4. **Home Screen Shortcut Icon**

   Shortcut home screen Icons are one of the features of native apps. Nonetheless, when the user visits a PWA using the same browser twice, the next time the client delivers a request to the user to install a shortcut home screen icon on his device [18]. If the user

accepts the request, a shortcut home screen icon will be installed in the user's device, which enables the user to access the application next time without the need to open a browser and write the URL of the web app.

5. **Native Look**

   Due to the possibility to add a shortcut home screen icon, the native look is another important feature of PWA. Tapping the shortcut home screen icon enables the application to launch in a full-screen standalone manner [10]. As a result, both the browser and the user's device UI will be hidden, and the client does not think that the application is working with the browser engine behind the scene [10].

Finally, according to Google documentation, for a web application to qualify the criteria of PWA, it should own a Web App Manifest, should be distributed over secure HTTP, should own a legitimate and registered Service Worker and should be visited at least twice within greater than five minutes. When the above criteria are satisfied, the browser will allow the user to download the app on the home screen for Android devices [10].

3. Building Blocks of Progressive Web Apps

This chapter evaluates the core building blocks of PWAs that are available in the browsers. However, before directly diving into the core features, a brief introduction to the evolution of the web and the distinction between PWAs and Single Page Applications (SPA) will be discussed in this section of the study.

3.1 Evolution of the Web and Single Page Applications

When the web was first invented by Tim Berners-Lee, most of the contents of the web pages were static texts and images. As a result, most web pages during the past had multiple pages and in most case these pages were static. Hence, traditional web pages were connected to each other using hyperlinks and each page refreshes when the user clicks on the navigation links to display their contents [19].

Nevertheless, based on the user's activity, server-side scripting languages enabled web servers to produce dynamic contents. Furthermore, the implementation of JavaScript in the browser in 1996 as a client-side scripting language eased the web development. JavaScript enabled web designers the possibility to include logic to ran in the client browser [19]. In 2005, a paradigm shift in the World Wide Web was observed when Asynchronous JavaScript and XML (AJAX) started to enable web pages to make asynchronous HTTP requests. Further, AJAX refreshes a portion of the web pages without stalling the User Interface (UI) and retaining the web page reactive.

The development of a new HyperText Markup Language (HTML5) standard began when the World Wide Web Consortium (W3C) and Web Hypertext Application Technology Working Group (WHATWG) started to work together around 2006. The HTML5 standard brought a revolution in the world of the web by adding several missing features to the web. Moreover, these new HTML5 features brought new techniques for developing a web application and enabled web developers to build better-off and multi-layered web applications [19].

As a result, the SPA is one of the new techniques for building a complex and multi-featured web application. As its name suggests, the SPA uses a single non-refreshing HTML page and relays on JavaScript for all user interactions and to communicate with the server, it uses AJAX [19].

Therefore, as related to the customary web application, the SPA provides a better user experience and fast server response time since server interaction is asynchronous, and it happens in the background. Furthermore, it transfers less data, portable and easier to distribute. Unlike native applications, which require administrative rights to install, SPA works by reloading the web page [20].

PWA is a modern philosophy for web design, which was launched in the year 2003 by Steve Champeon. The main idea behind progressive enhancement is that Web pages should be first developed starting from the less feature-rich browsers and then adding more advanced features to enable them to be capable of modern browsers. It follows bottom-up philosophy, unlike elegant degradation that proposes developing web page top-down [21]. However, there is a lot of misunderstanding among users and new developers regarding the concept of SPAs and PWAs. Figure 3.1 below shows a brief distinction between a single page application (SPA) and PWA.

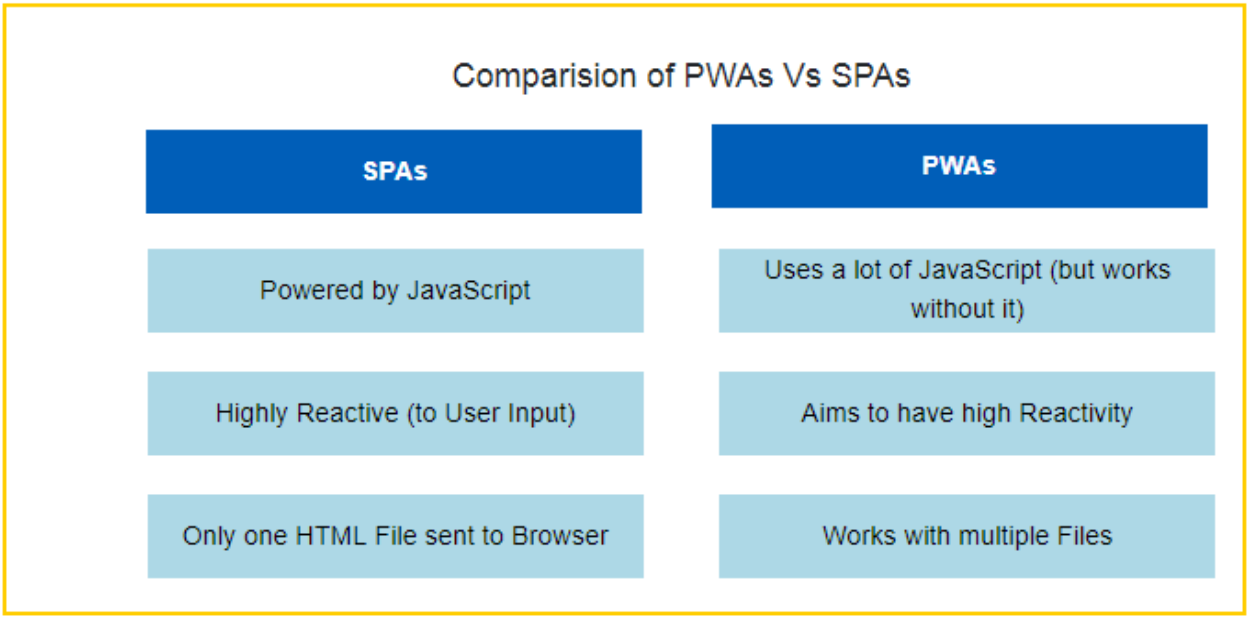

Figure 3.1: Brief distinction between PWAs and SPAs

As shown in the figure, the main difference between the two is that PWA works with or without JavaScript and with multiple files, whereas SPA depends heavily on JavaScript, highly reactive to user input and send only one HTML file to the browser.

3.2 Core Building Blocks of Progressive Web Apps

The core building blocks of PWAs are represented in figure 3.2 below.

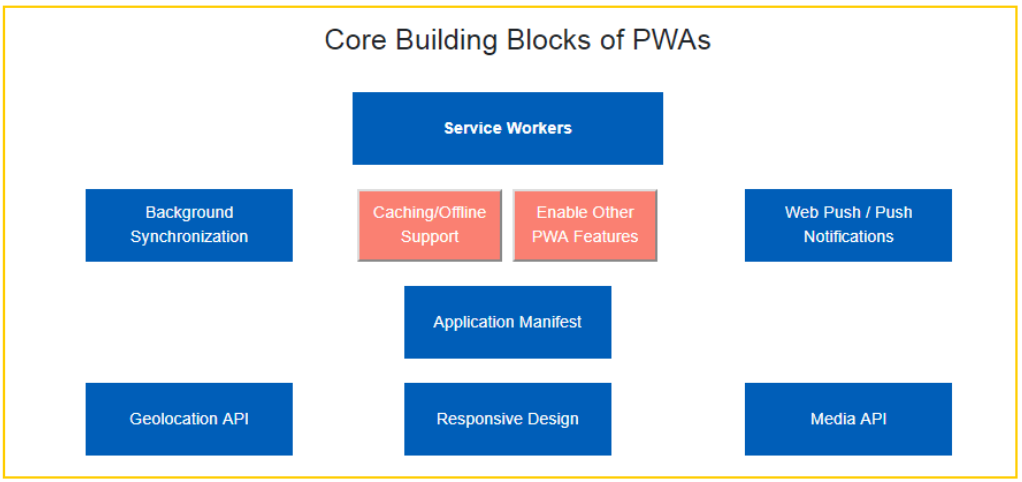

Figure 3.2: Core building blocks of PWAs

As demonstrated in figure 3.2, the core building blocks of PWAs are Service Worker, Background Synchronization, Web Push Notifications, Geo-Location API, Responsive Design and Media API [6]. In addition, IndexedDB will also be used to store data for offline use. In the next section of this study, each technology except responsive design will be elaborated in more detail.

3.2.1 Web App Manifest

The Manifest file is one of the core building features of a PWAs, which enables enormous information about the app. It gives information regarding the app like name, author, icon, description, etc. It enables the app to be installable by providing the home screen icon so that it feels and behaves like a native app [22]. Tapping the home screen icon directly launches the web application, which minimizes the traditional way of opening the browser and writing the application's URL address on the browser. As a result, it allows users to interact easily with the application.

The manifest file lists all the required icons for different devices. It is a JavaScript Object Notation (JSON) file, which lists different properties about the application. Hence, browsers use

the best fitting properties from the file to provide richer user experiences (UX) like offline functionalities. Listing 3.1 below shows the properties of a sample manifest file.

Manifest Properties

JSON File

```
{
  "name": "Programme management annual clock", //Long name of the app (e.g. on Splashscreen)
  "short_name": "annualclock", //Short name of the app (e.g. below icon)
  "start_url": "/index.html", //Which page to load on startup
  "scope": ".", //Which page to include in "PWA Experience"
  "display": "standalone", //Should it look a standalon app?
  "icons": [...], //list of configure icons (e.g. on homescreen)
  "theme_color": "#000000", //Theme color (e.g. on top bar in task switch)
  "background_color": "#FFFFFF", //Background whilst loading & on SplashScreen
  "orientation": "portrait", //Set (and enforce) default orientation
  "dir": "ltr", //read direction of your app
  "lang": "en-US", //Main language of app
  "description": "Keep running ..." //Description (e.g. as favorite)
}
[
  "related_applications": [{
    "platform": "play",
    "url": "https://play.google.com/store/apps/details?id=com.example.app1",
    "id": "com.example.app1"
  }]
]
```

Listing 3.1: Screenshot of a Manifest file

As shown in the listing, it is a JSON file, which lists info regarding the app and passes that information into the browser. The browser then uses that information to display the app to the user and to install a home screen icon. The purpose of each key-value pairs is explained to the right side. Figure 3.3 below shows browsers that fully (green) and partially (yellow) support Web App Manifest.

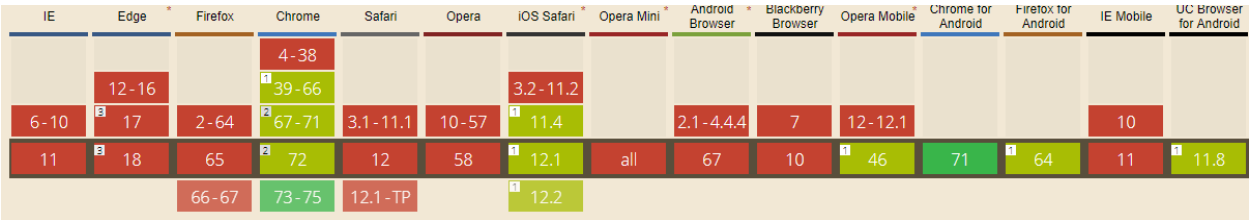

Figure 3.3: Browser Compatibility of Web Manifest App [22]

As illustrated in the above screenshot, to date, only Chrome version 73-75 and Chrome for Android version 71 fully support Web App Manifest. Other browsers such as Chrome versions 39-72, iOS Safari versions 11.4, 12.1 and 12.2, Opera Mobile version 46, Firefox for Android version 64, Samsung Internet, UC Browser for Android, QQ browser and Baidu Browser partially support the Web App Manifest API and it is under development in other browsers(red) [22].

3.2.2 Service Workers

One of the core building blocks that enable to build PWAs is a Service Worker. Without Service Workers, developing a PWA with offline functionality, Push Notification and Background Synchronization is impossible [24]. It is a script that runs in the background. Web app will be used to install Service Worker into the browser [24]. Like a Web Worker, Service Worker runs in a separate thread and does not access the Document Object Model (DOM) directly. It is a programmable network proxy, which acts as a liaison for every user request and manages those requests in a web application [24]. Figure 3.4 below depicts the difference between JavaScript and Service Worker.

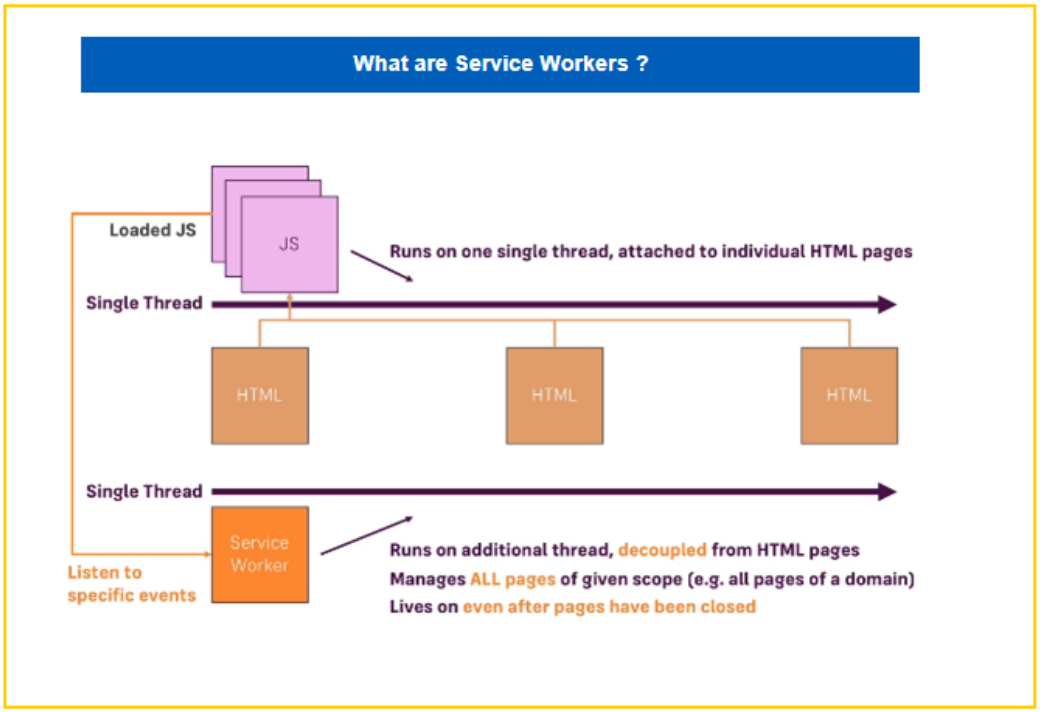

Figure 3.4: Difference between JavaScript and Service Worker

To install a Service Worker, the application developer initiates a request by writing a special script, which will be written with JavaScript [24]. The script will initially help to download those static assets, which are required to be available when the user is offline. During installation, failure in any of these static assets' results termination of the previous installation, and a new installation will be retired when the consumer uses the app another time. Hence, the Service Worker will be triggered when the client uses the app following the first installation [24].

Regarding the network connection, the Service Worker requires a secure connection to reduce the impact of interfering network requests. Moreover, using HyperText Transfer Protocol Secure (HTTPS) helps to minimize possible attacks like hijacking a connection and fabricating a response with a man-in-the-middle [24]. However, during development, it is possible to use localhost with a Service Worker [24].

Regarding events, Service Workers have a bunch of listenable events. Some of the most known listenable events that are available in Service Workers are: fetch event, push notification event, notification interaction event, background sync event, and service life cycle event. The Fetch event is an HTTP request event, which is initiated by the client or page-related JavaScript. Furthermore, Service Worker receives Web Push Notifications from the server and the user interacts with displayed notification. In addition, Service Worker receives Background Sync event for instance when an internet connection is restored. Finally, Service worker has an event, which changes the life cycle of the Service Worker.

The function of Service Worker is to progressively enhance the application, which improves the contents as well as the functionality of the application. It can be added to the new or existing fully functional applications. However, in order to get the best out of this technology, the user needs to use the application with Service Worker compatible browsers [25]. Figure 3.5 below represents the browsers compatibility of Service Workers.

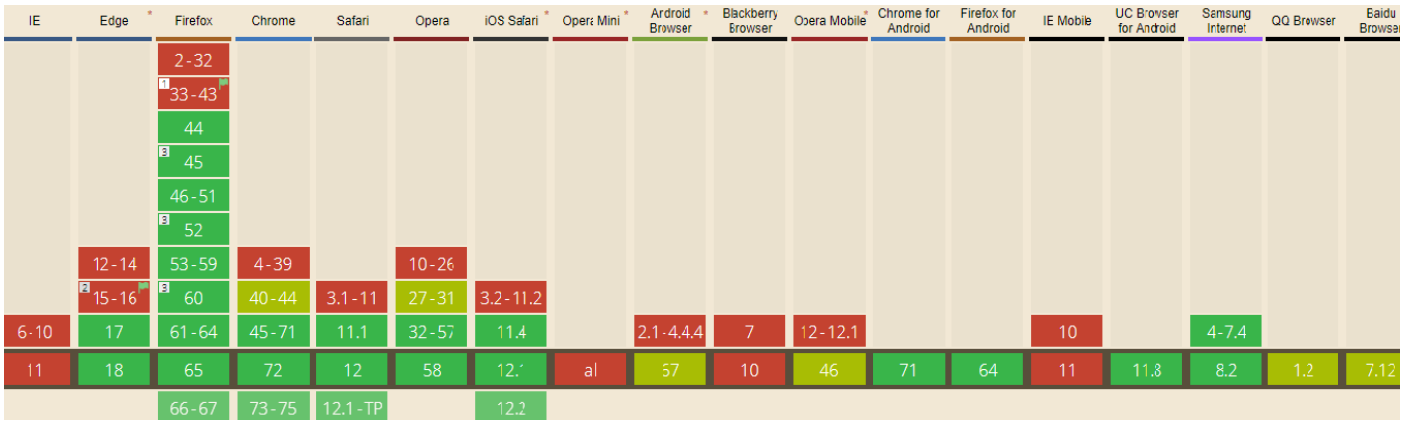

Figure 3.5: Compatibility of Service Workers with major browsers. [25]

Note: The numbers depict browser versions, Green shows full support, yellow shows partial support and red shows that the browser version does not support Service Worker

As illustrated in the figure, up till now, not all browsers support the Service Worker, but most recent versions of the major browsers, such as Firefox, Chrome, iOS Safari, Internet Edge, Opera and Safari are compatible with the Service Worker and it is under development in other browsers.

3.2.2.1 The lifecycle of Service Workers

As stated in this study, Service Worker is a script that is driven by an event. It has many listenable events, of which lifecycle event is one of those that changes the life cycle of the Service Worker. As a result, it runs only if it receives the events. Using Service Workers, developers treat the network as an enhancement and Service Worker enables to cache static assets. Hence, the proper way of caching static resources in the browser helps to utilize the offline functionality of the Service Worker. Saving static assets in the browser minimizes the hurdle of network and increases the performance of the application by enabling it to load faster and reducing the back trip between the client and the server [25].

The life cycle of the Service Worker is totally separated from the web page. Figure 3.6 below describes the lifecycle of Service Worker.

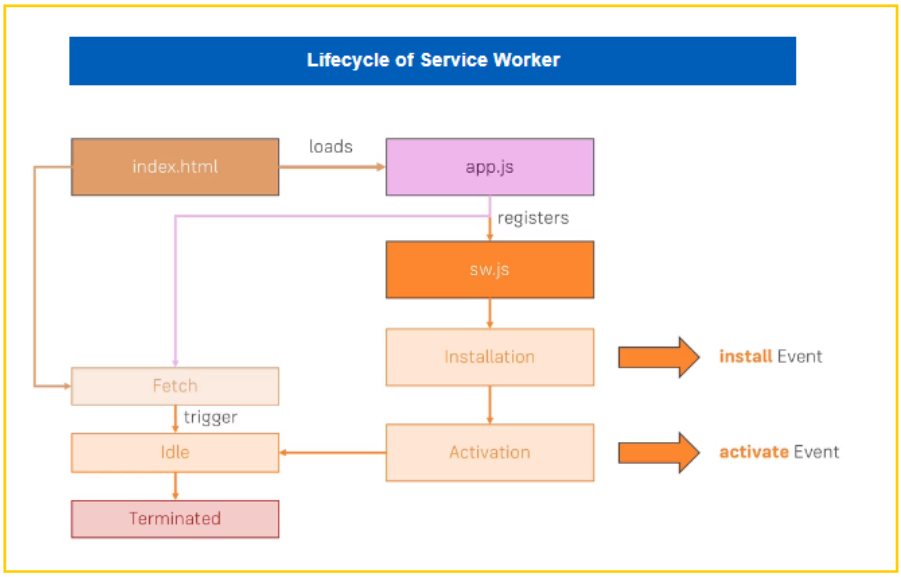

Figure 3.6: The Lifecycle of Service Worker.

The above figure illustrates how Service Worker is registered. Like any traditional web application, first, the index.html file loads the app.js file. In the app.js file, to register a Service Worker the developer writes some code and the code exists in a separate JavaScript file. The purpose of the code was to let the client that the sw.js file is a JavaScript, but does not run immediately as it can be done in other JavaScript files rather registers as Service Worker in the background. However, to register the Service worker, first, there is a code inside sw.js, which checks whether the browser supports service worker or not. Listing 3.2 below illustrates the code to check browser support for a Service Worker.

```
//browser support for serviceWorker
if ("serviceWorker" in navigator) {
  window.addEventListener("load", function() {
    navigator.serviceWorker
      .register("/sw.js")
      .then(function(registration) {
        //Registration was successful
        console.log(
          "serviceWorker registrattion successful with scope:",
          registration.scope
        );
      })
      .catch(function(err) {
        //Registrtion failed
        console.log("serviceWorker registration failed:", err);
      });
  });
}
```

Listing 3.2: Code to check browser support for Service Worker

Based on the code shown in the above listing, the developer checks whether the browser supports Service Worker or not. If the browser supports the API, it registers the sw.js file when the page loadeds for the first time [24]. And due to that registration two lifecycle phases are reached. First, the Service Worker will be installed by the client browser and as a result releases an install event, which can evoke to carry out some code inside of the sw.js file and later used to cache static resources. The browser installs a new Service Worker only if there is an update in the existing files or if it is the first time not in every page refresh.

Activation event is another event, which is executed as soon as the installation is finished. But not activated immediately instead will be activated depending on the availability of an old version of the Service Worker. If the version of Service Worker running is not old, then it is activated otherwise, closing the existing browser tabs and reopening a new tab installs a new Service Worker. Closing an existing browser tab and reopening a new tab is required to activate the Service Worker since it is not attached to a single page, but to the overall domain or scope and lives, even after the browser is closed. Hence, doing so fires the activate event. After activating the Service Worker, all pages of the given scope will be controlled by the Service

Worker. Figure 3.7 below demonstrates a screenshot of registered and an activated service worker running on Chrome dev tools.

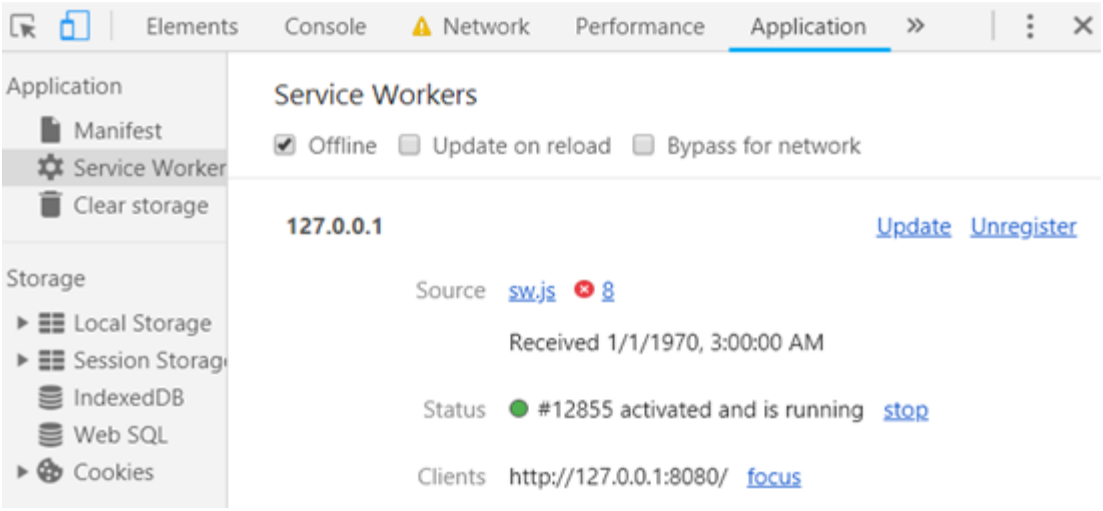

Figure 3.7: Registered and Activated Service Worker inside Chrome Dev Tools

Once the Service Worker activated, it enters an idle mode, which means basically it sits there since Service Worker is a background process handling events and if there is no event coming, it does nothing. Following that, after a time of idling around, it will terminate, which does not mean that it is uninstalled or unregistered, but it means it is on sleep mode. A sleeping Service Worker can waken automatically as soon as events are coming in, for instance, a fetch event and all other events. If a fetch event occurs either due to an HTML file requesting a resource or the developer sending a fetch request in the JavaScript code, the Service Worker is woken up and trigger a fetch event and then put back to idle mode.

3.2.2.2 Caching with Service Workers

The main purpose why we are using Service Workers is to provide offline support for our Web application. That means users can use the web application even if they are offline. However, this raises questions like why this would be beneficial? When would the user use a web app if no internet connection? Figure 3.8 below shows the use cases to answer these questions:

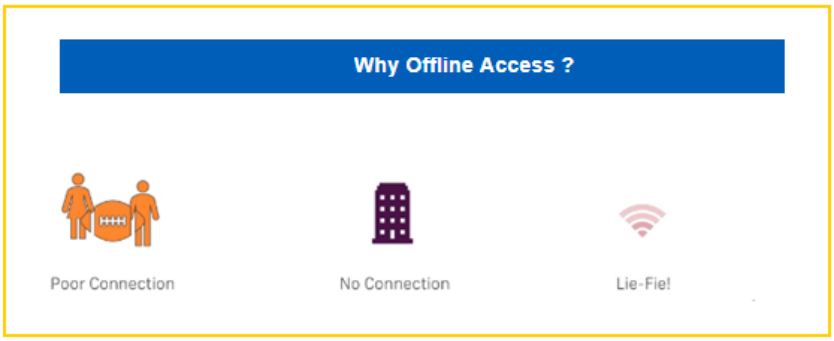

Figure 3.8: Use cases to provide offline support

Figure 3.8 illustrates examples when web app users require offline support. There are situations where a large crowd of people gathers together for a sports event and due to that there is a poor connection or no connection, but the users want to read a news article or to navigate between pages, but cannot do so due to the poor connection. However, with Service Workers, the developer can precache or cache certain files and assets to the web app in order to enable the web app to display them in a situation when the app is offline. Hence, in such circumstances, providing offline functionality to an app is super important. In addition, another example may be inside an elevator the user may be out of connection for a limited time and still, want to access the app. Finally, and probably the most important use case is Lie-Fie that is WIFI which is not. Application users think that they have WIFI and want to access the application, but it is not. Hence, using the Service Worker, the developer can cache certain resources to display when the application is offline.

Developers use the cache API to access the web application when the app user has no network access. When working with a web application, caching can be done either in the server or in the browser. In order to manage the caching made in browser, developers use cache API, by storing data as a key-value pair when there is the connection and fetching them when the application is offline. These can be done either using Service Worker or ordinary JavaScript. Figure 3.9 below shows how the cache API works.

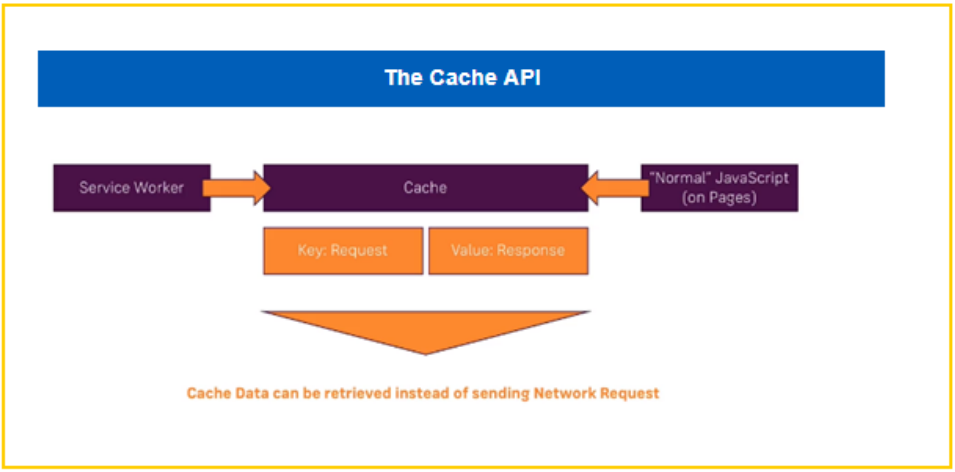

Figure 3.9: How the Cache API works

Cache API is supported by most major browsers. Figure 14 depicts the list of browsers that are compatible with Cache API. As shown in the figure, most except for Internet Explorer and Safari versions less than 11 support the API. In addition, Edge mobile and Safari on iOS are not compatible with mobile phones.

Browser compatibility

Update compatibility data on GitHub

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Edge Mobile	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet
Basic support	43 *	Yes	39 *	No	30 *	No	43 *	43 *	No	39	30 *	No	4.0
add	44 *	16	39 *	No	31 *	11	44 *	44 *	No	39	31 *	No	4.0
addAll	46 *	16	39 *	No	33 *	11	46 *	46 *	No	39	33 *	No	5.0
delete	43	16	39 *	No	30	11	43	43	No	39	30	No	4.0
keys	43	16	39 *	No	30	11	43	43	No	39	30	No	4.0
match	43	16	39 *	No	30	11	43	43	No	39	30	No	4.0
matchAll	47	16	39 *	No	34 *	11	47	47	No	39	34	No	5.0
put	43 *	16	39 *	No	30 *	11	43 *	43 *	No	39	30 *	No	4.0

Full support — No support

Experimental. Expect behavior to change in the future. * See implementation notes.

Figure 3.10: Screenshot of the list of browsers that support Cache API

3.2.2.3 Static caching of the Service Worker

To install a Service Worker, the developer should release a new version, or the file should be changed otherwise it will not get replaced. For recent installation phase of Service Worker is a great place to cache some assets, which do not change that often like app shell, the toolbar, basic styling that might not change every second. Scripts, which will be changed during deployment will be updated but does not change that often or created by the user dynamically. Hence, to update these scripts we use Cache API. Figure 3.11 below depicts static caching at installation.

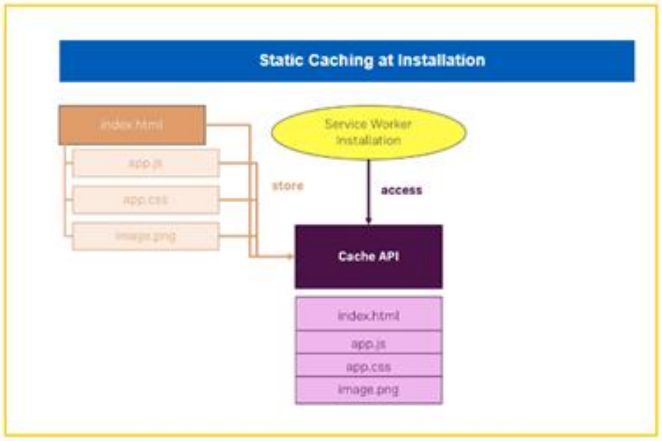

Figure 3.11: Static caching at Installation

As illustrated in the above figure, for a given route, for example, the developer uses index.html and inside that file, the app loads a couple of assets like main scripts, main stylesheet, and images. Then, the application caches these files during Service Worker installation, which allow developers to get access to Cache API and store these files in the cache. These files will be available on the next page visit of the user and later fetched even if the user does not have access to the internet.

The following listing shows the code to get access to Cache API and to cache the file app.js:

```
self.addEventListener('install', function(event) {
  console.log('[Service Worker] Installing Service Worker ...', event);
  event.waitUntil(
    caches.open('static')
      .then(function(cache) {
        console.log('[Service Worker] Precaching App Shell');
        cache.add('/src/js/app.js')
      })
  )
});
```

Listing 3.3: Code to get access to the cache API

However, though the app.js file is cached in cache API, the app users cannot access the application when it loads while they are in offline mode. The reason for this is that the application is cached, but the developer did not fetch it to retrieve the item from the cache. The following listing allows fetching the cached assets in offline mode.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request)
      .then(function(response) {
        if (response) {
          return response;
        } else {
          return fetch(event.request);
        }
      })
  );
});
```

Listing 3.4: Code to Fetch cached assets in offline mode

However, to add all assets in the cache, the developer needs to use the following listing:

```
self.addEventListener('install', function(event) {
  console.log('[Service Worker] Installing Service Worker ...', event);
  event.waitUntil(
    caches.open('static')
      .then(function(cache) {
        console.log('[Service Worker] Precaching App Shell');
        cache.addAll([
          '/',
          '/index.html',
          '/src/js/app.js',
          '/src/js/feed.js',
          '/src/js/promise.js',
          '/src/js/fetch.js',
          '/src/js/material.min.js',
          '/src/css/app.css',
          '/src/css/feed.css',
          '/src/images/main-image.jpg',
          'https://fonts.googleapis.com/css?family=Roboto:400,700',
          'https://fonts.googleapis.com/icon?family=Material+Icons',
          'https://cdnjs.cloudflare.com/ajax/libs/material-design-lite/1.3.0/material.indigo-pink.min.css'
        ]);
      })
  )
});
```

Listing 3.5: Adding all assets in the cache to access them in offline mode

After caching the above files, reopening a new tab in the browser and refreshing the Service Worker, if the developer selects the offline mode in the network tab and refresh the tab, the following files will be seen:

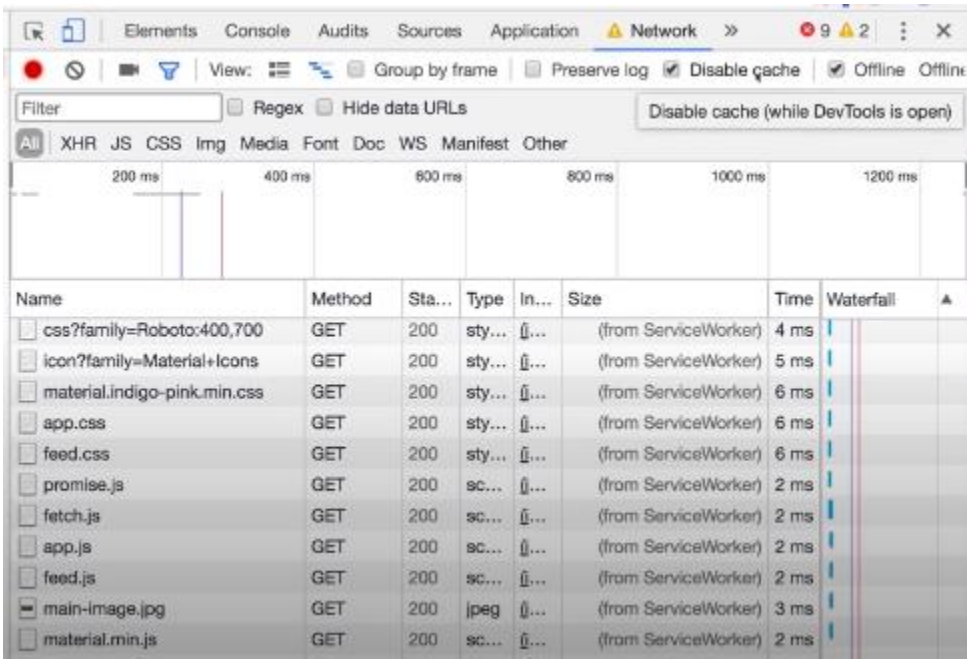

Figure 3.12: Screenshot of the Chrome network tab

As shown in figure 3.12, the disable cache and offline checkboxes are checked. After that, refreshing the network window retrieves those assets that are cached. As illustrated in the figure, these resources are not from the network, instead they are from the Service Worker and hence it enables to access the application offline.

The caching strategy followed above is static caching and it will be installed when the Service Worker is installed. However, static caching does not update files once it is installed unless some changes are made in those files, which are installed. Hence, in order to solve this problem, the developer needs dynamic caching upon fetching by providing different version numbers to caches.

3.2.2.4 Different Caching Strategies with Service Workers

The caching strategy used with fetch event listener is cache with network fallback. The way it works is that first, it tries to access the cache and if that fails or if it fails to find the item in the cache, it fallbacks to the network. Figure 3.13 below illustrates the way cache with network fallback works.

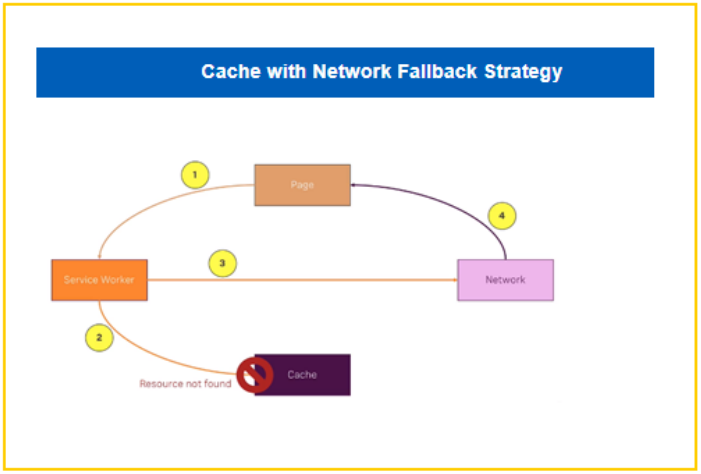

Figure 3.13: Cache with network fallback strategy

As shown above, in the figure, the Cache with network fallback strategy works first the page sends a request to the Service Worker or in another word the later interrupts any request sent by the page. Following that the Service Worker looks at the cache and if the requested resource

is found in the cache then it returns and if it is not found, it executes another step, it gets out to the network so that it responds accordingly. That is why it is called cache and then network fallback. The good side of this strategy is that it can instantly load assets if it has them in the cache even if there is network, but the bad thing about this strategy is that it returns old version of the resources, which are in the cache because, it is difficult to reach out the network by default. Another simple strategy is the cache only strategy. Figure 3.14 below depicts how this strategy works.

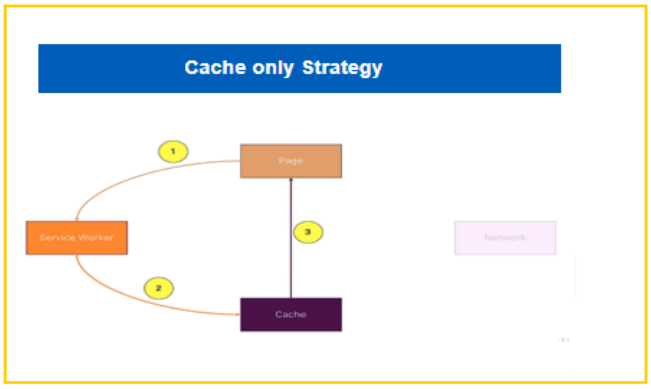

Figure 3.14: Cache only strategy

As shown above, in the figure, the cache only strategy first reaches out the Service Worker or in other word, the Service Worker intercepts the request. Then the application looks at the cache and if it finds the resource there, it returns to the page totally ignoring the network. The opposite of the cache only strategy is a network only strategy. Figure 3.15 below shows how the network only strategy works.

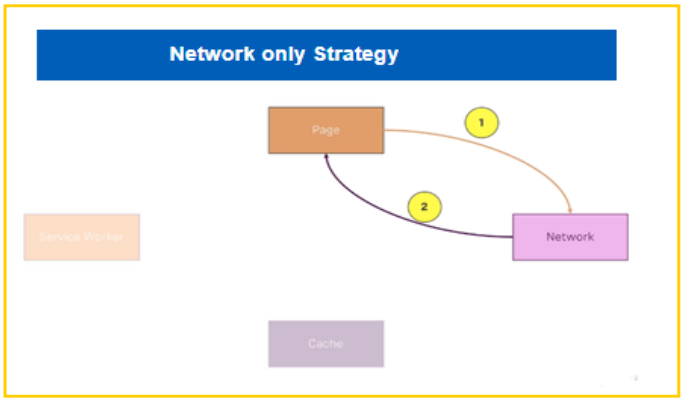

Figure 3.15: Network only strategy

As illustrated in the figure, in the network only strategy, the application does not use Service Worker at all instead the page sends a request to the network and return that. However, this strategy does not make any sense since it does not use any cache event and make use of only the network.

Another interesting cache strategy is the network with cache fallback. Figure 3.16 below depicts the way this strategy works.

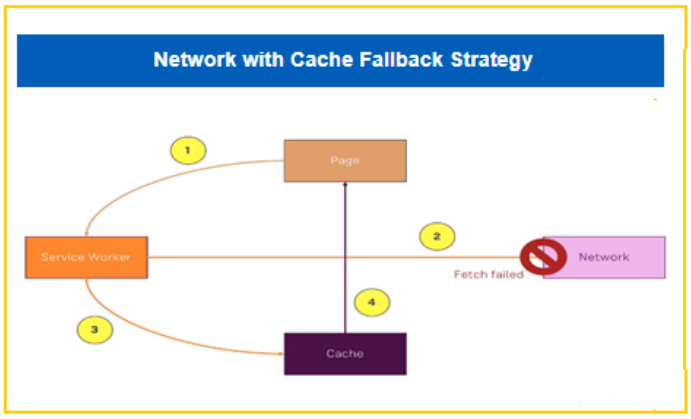

Figure 3.16: Network with Cache fallback strategy

As shown in the figure, this strategy makes a lot of sense. First, the page catches a request to the Service Worker and then try to reach out to the network. If that fetch to the network fails, only then it reaches out to the cache and returns the assets from there. Otherwise, if the network request would have been successful, then it would have been returned the network response and ignore the cache. This strategy makes sense because it makes use of the best out of both, cache and network. However, the downside of this strategy is that it does not use the fast assets that are stored in the cache, which is good even if we have an internet connection. Finally, in order to solve the downside of this strategy, we need to follow the cache then network strategy. Figure 3.17 below depicts how the cache then network strategy works.

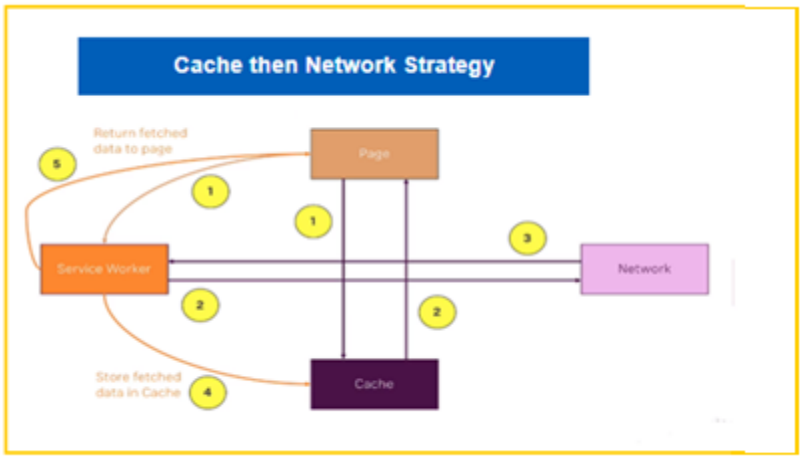

Figure 3.17: Cache then network strategy

As illustrated in the figure, the first step in this strategy is that the page directly accesses the cache and gets the value back without the involvement of the Service Worker. However, at the same time, the page also reaches out the Service Worker. Then, the Service Worker goes to the network to get a response from there. The network response goes back to the Service Worker and then store the fetched response in the cache and return the fetched data to the page.

3.2.3 IndexedDB

It is a key-value pair and transactional database operating in the browser [27]. It is transactional because the failure of a single action within a given transactions affects all actions of that transaction to maintain the integrity of the database. In addition, it allows storing substantial volumes of unstructured data containing blobs and files [27]. It is unstructured data because it does not require to define a schema upfront. Instead, we can store whichever data on it. IndexedDB is best for JSON data but theoretically, we can store files and data on it. Due to its key-value pair nature, it is basically like a JavaScript object, nested objects or arrays [23]. Another important feature of IndexedDB is that it can also be accessed asynchronously. Accessing IndexedDB asynchronously is important because we can use it in the Service Worker, since Service Worker is all about listening events and most of these events have asynchronous nature and hence, they can be run in the Service Workers.

IndexedDB can contain more than one database but one per application. Nonetheless, in the given database, it is possible to store multiple objects like a table and inside the object store, we can store the object. IndexedDB has good browser support [28]. Figure 3.18 shows a screenshot of the browsers that support IndexedDB.

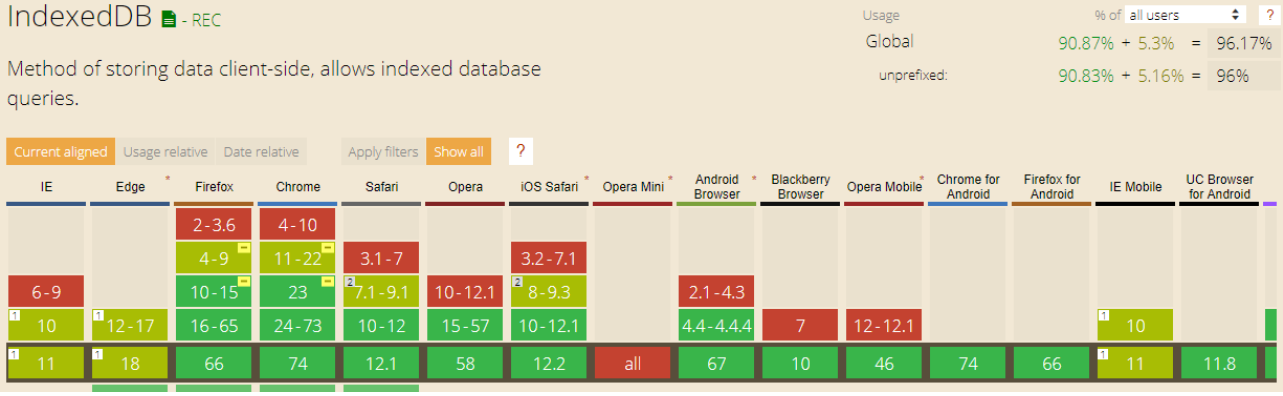

Figure 3.18: A screenshot of IndexedDB browser support. [28]

As shown in the figure, IndexedDB has globally 96.17% browser support.

3.2.4 Background Synchronization

One of the new technologies that can be used in developing a PWA is background synchronization. It allows synchronizing data even when the user is offline when sending the data [32]. It is a situation when we have developed an offline ready application and the user accesses and navigates the app while offline. With PWA, the user can store the request they want to send in the application while accessing offline, and the data will be sent later when connectivity is reestablished [32]. Figure 3.19 below depicts how background synchronization works.

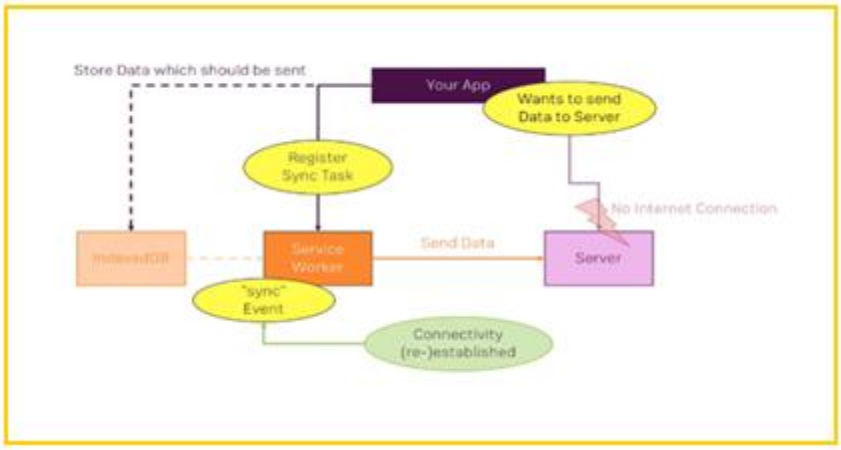

Figure 3.19: Screenshot of how background synchronization works

As described above, background sync is sending data to the server when the internet connection is down. As illustrated in the figure, the web application is running in the browser and a backend server, which is serving the HTML page, or the API building. The goal is to send the data to the server while the user is offline. Sending the request fails because of lack of internet connection, but the request will be saved by the application to be processed later when connectivity is established.

With Service Worker, the application cannot cache post requests to be sent some time in the future, but the app can cache the response. But, there is a technic to register sync task on the Service Worker and store the data to be sent with request inside the IndexedDB. Eventually, when connectivity is reestablished, the Service Worker goes ahead and immediately execute the sync event, which let the Service Worker fetch the data from the IndexedDB and send to the server. The good thing about this process is that background synchronization works even though we close the browser tab or the device.

SyncManager is the API that helps to use background synchronization features [30]. Figure 3.20 below shows the list of browsers that support SyncManager API as of to date.

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Edge Mobile	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet
SyncManager	49	No	No	No	No	No	49	49	No	No	No	No	No
Available in workers	61	No	No	No	No	No	61	61	No	No	No	No	No
register	49	No	No	No	No	No	49	49	No	No	No	No	No
getTags	49	No	No	No	No	No	49	49	No	No	No	No	No

Figure 3.20: browser compatibility of SyncManager API. [30]

Regarding browser compatibility, as displayed in the figure above, to date only chrome version 49 and 61 both for desktop and android supports SyncManager API. Hence, during development, we need to check whether SyncManager is in the window or not in order to use the API. The following listing illustrates the code to do this:

```
if ('serviceWorker' in navigator && 'SyncManager' in window) {
  navigator.serviceWorker.ready
    .then(function(sw) {
      var post = {
        id: new Date().toISOString(),
        title: titleInput.value,
        location: locationInput.value
      };
      writeData('sync-posts', post)
        .then(function() {
          return sw.sync.register('sync-new-post');
        })
        .then(function() {
          var snackbackContainer = document.querySelector('#confirmation-toast');
          var data = {message: 'Your Post was saved for syncing!'};
          snackbackContainer.MaterialSnackback.showSnackbar(data);
        })
        .catch(function(err) {
          console.log(err);
        });
    });
} else {
  sendData();
}
});
```

Listing 3.6: Code to check whether SyncManager is in browser or not

3.2.5 Web Push Notifications

It is one of the best features that native apps can use, and access. Web Push Notifications is the best way of getting users back to our application. Assume that the user has a mobile phone in his pocket and suddenly it vibrates and when he looks at it, the user sees a new article, which he needs to read in his favorite app. Then, the user taps on the notification and opens the article. From the developer perspective, it is the best way to drive user engagement and re-engage users into the application [33]. On the contrary, from the client's point of view, they will be get informed about things, which matters to them at least if it is done in a good way.

Web Push Notification is an extremely powerful feature, which the developer can add to PWAs to really turn it to native app experiences. It notifies web apps about important events. Why push notifications? There are many reasons, but the following are the main ones:

- Show up important events if the app (and browser) is closed
- Drive user engagement [33]
- Mobile-App like experience

Figure 3.21 below illustrates how push notifications work in the web app.

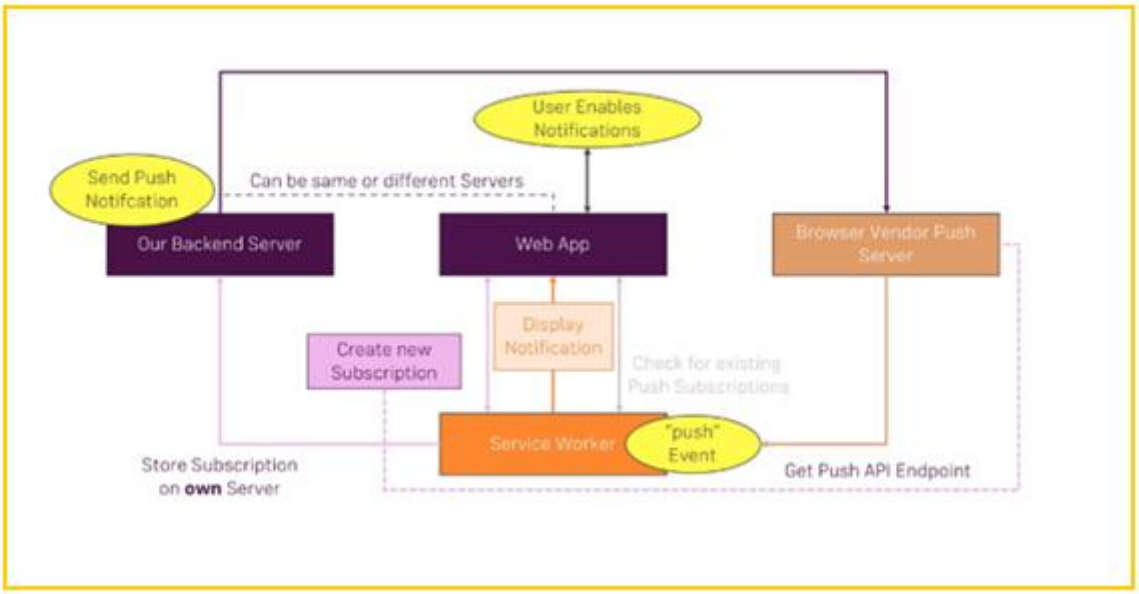

Figure 3.21: How push notification works

As shown in the figure, assuming the web app has a Service Worker running in the background, push notifications contain two parts [33]:

- **Push** - which is some information pushed by the server to our applications and
- **Notifications** – information that is shown to the user

As illustrated in the diagram, first the app requests the user to enable notifications otherwise it is difficult to show a notification. Allowing permission is a onetime task though the user can remove it later and once permission is given, we can start displaying notifications, which are independent of pushing messages. These notifications are triggered by JavaScript, which triggers users to subscribe to our service by informing all users about a new post. In a web app, we can check for an existing Web push subscription, which is created for each browser in a given device. Hence, a subscription is a given browser device combination and different browser vendor owns push notification server. The following figure shows displaying notifications in a web app.

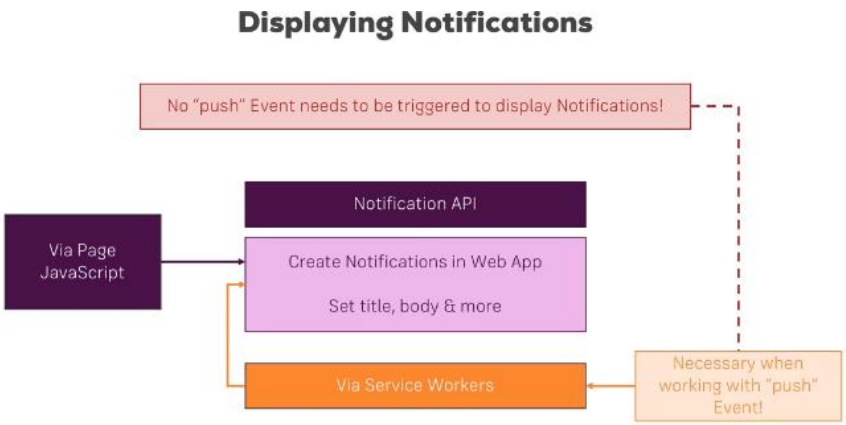

Figure 3.22: Screenshot showing how to display notifications

Regarding browser compatibility of push notifications, figure 3.23 below lists the browsers that support the API.

Browser compatibility

Update compatibility data on GitHub

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Edge Mobile	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet
`Notification`	22 ★	14	22	No	25	6	No	Yes	?	22	Yes	No	Yes
Available in workers	45	Yes	41	No	32	?	No	45	Yes	41	32	No	?
Secure context required	62	?	67	No	49	?	No	62	?	67	46	No	?
`Notification()` constructor	22	Yes	22	No	25	6	No	Yes	?	22	Yes	No	?

Figure 3.23: Push Notifications browser compatibility. [31]

As illustrated in the table, the Notification API is supported by most known browsers. Only Internet Explorer for desktop and Safari iOS and Android WebView in mobile do not support the API.

3.2.6 Native Device Features

It includes accessing device camera and geolocations. Previously, these features were basically only available in native applications, in iOS and Android. However, with PWAs, these features can be accessed through the web. These two features give users a look and feel that can only match native experience. For instance, by adding geolocation capability in the web application, it is possible to know and analyze where the clients of the application reside or from where access the web to offer them a better service. Moreover, geolocation is also used to track up-to-date local information, Global Positioning System (GPS) navigation to guide app users to a specific location or show points of interest. Nonetheless, getting the geographical position of the user can compromise the privacy of the user and thus, it is impossible to access their location unless the users approve it. Most browsers support geolocation and accessing device camera though some require a few configurations to make it work.

4. Requirement Analysis and Design

In this part of the study, I will present an overview of the requirement analysis and design of the annual clock app. As its name illustrates, the application is planned to be used to manage the annual tasks of Aalto University. The first section of this chapter presents the requirements defined to develop the app. Following that, an overview of the used technologies will be presented in the technologies and concepts section of the study.

4.1 Requirement specification of the annual clock

The programme director must be aware of and lead several processes in the program: marketing of the programme, student admissions, curriculum design, continuous evaluation and development of education. The aim of the annual clock is to help the programme director and other people working with the programme, and students in the programme, to understand these processes.

The starting view of the annual clock shows all the information in the clock. The user can filter the view by selecting the processes, the level of the degree programme (bachelor's or master's programme) and the schools. Using the clock happens by utilizing the filtering: all processes about the degree programme contain so much information that it is not possible to review it all at once.

Each process consists of several events. Each is shown on own layer and they are sorted primarily according to the start time and secondarily by the end time. Each event has the following information: name of the event, description, start time and end time, the school, and the level of the degree programme. For each event, it is also possible to link to a web page for more information on the topic. Color for displaying the event in the annual clock is also defined. Usually, all the event of a certain process has the same color, but it is possible also to represent the events in different color tones.

Regarding the admin interface, it is possible for an administrator to add process events and event-related descriptions, start and end times, and links to the management interface side. Also, it determines the level of the degree program and school the event is associated with, as

well as determine the color of the event. After the admin user has saved the event in the annual clock it will be shown in the clock.

Concerning the layout of the application, there are two academic years (from the beginning of August to the end of July) are displayed in the clock at once, as some of Aalto University's processes have a two-year cycle. The clock can be viewed as either a semi-circle or a full circle. The colors of Aalto University are utilized in the processes of the clock. On a narrow screen, the clock filter function can be opened and closed by pressing a button. On a narrow screen, the two years of the clock are shown one below the other.

Generally, the requirements for the annual clock application are listed below:

- Register and login as an admin
    - Though the application has a register and login functionality, the user is not required to register into the system to view the application. However, the administrator needs to register in order to perform CRUD (create, read, update, delete) operation.
- Create a task
    - The admin creates tasks so that the user views them in the front page
- Edit a task
    - The admin can rename tasks in order to fix any mistakes
- Delete a task
    - The admin can remove tasks in case they added them by mistake or do not need them anymore

Furthermore, as explained in the literature review part of the study, the application is implemented on the offline-first concept and includes many PWAs features. Hence, some of the requirements identified for an offline-first web apps are listed below:

- Access the application offline
    - The app user can view the application offline in situations when they suddenly lost internet connection, but decided to use the application
- Create or edit tasks offline

    - The admin can add or make changes to existing tasks while the app is offline and continue working irrespective of their network status
- Background Synchronization
    - The application must synchronize the tasks added or edited while the application was offline when the user gets connected to the network.
- Save data on IndexedDB
    - The application data were saved on the browser's IndexedDB so that they can be accessed offline
- Optimistic UI
    - Since the application data were saved on IndexedDB, they must load fast regardless of the internet connection
- Push Notifications
    - When the user is subscribed to push notifications, the app sends a push message when new data is added
- Accessing device camera
    - Possible to capture profile image if the admin user wants to change the avatar
- Browser support
    - The offline functionality of the application works with the latest versions of most web browsers that are enabled with Service Workers.

4.2 Technologies and Concepts

This section of the study presents an overview of the technologies used to develop the task management annual clock application. The application consists of two components: frontend and backend. The frontend is developed as a SPA and it is based on offline-first concept mentioned in the previous section of the study. React.js is used to write to the client code, which was first introduced by Facebook in the year 2013 and made open source in 2015 [34]. To implement the backend of the application, Node.js is used. In the following section of this study, I will briefly discuss these two technologies.

4.2.1 React.js

It is a component-based JavaScript library. It is used to develop an interactive user interface (UI). According to the official React.js documentation, React.js is a library for developing flexible

UIs [36]. It is the most common frontend JavaScript library at present and includes the view (V) layer in the model view controller (MVC) pattern. DOM is composed of HTML template, retrieved from various files and scripts [37]. React.js supported by a group of different developers and organizations and by most popular social media like Facebook and Instagram [38]. The main target of React.js is to provide better UX by developing robust and fast web apps [38]. It is also possible to integrate React.js with other JavaScript frameworks or libraries.

For manipulating the UI, it uses and keeps track of the virtual Document Object Model (DOM), which makes the site more interactive and increase the performance of the app [34]. Virtual DOM is the abstracted representation of the real DOM and it uses events to send simple commands like create, update, delete to build a simple, robust and performing applications [35]. For most front-end applications, synchronizing the data model with the DOM was the major source of bugs [35]. The main reason Facebook developed React.js is to solve the problem arising from developing a large application that often uses changing data without subsequent page refreshes. Hence, the strategy React follows is Component-Based, declarative, Learn Once and Write Anywhere. It is simple to generate an interactive UIs with React, when the data changes, it updates and renders the required components and its Views are declarative which makes easy to predict and debug [34].

As stated above, the structure React.js adopts is component-based and instead of using templates, JavaScript is used to write this logic. Writing components in JavaScript creates a simple route of rich data across the web application and maintaining the state away the DOM. Most of the time React components are reusable and help to solve the problems arising from developing the app. Furthermore, when the modules get complicated, they are divided into simpler and smaller ones and in most cases, these React components are like JavaScript function [34].

Another important feature of React is that, using NodeJS it can be mostly rendered in the server side and using react Native, it supports mobile applications. Furthermore, it implements one-way data flow simplifying the boilerplate and easing the traditional data binding [34]. As a result of this, when related to other libraries and frameworks, React.js is the most loved among developers. React exceeded its competitors such as Ember and Angular JS after three years of

its announcement. Currently, in the world of frontend web development, React.js is considered as a standard [34].

According to a survey made by Stack Overflow, for seven years in a row and in 2019, the most frequently utilized programming language is JavaScript and the most loved and wanted web frameworks by developers in 2019 is React.js followed by Vue.js [38]. Figure 4.1 shows the list of most loved, dreaded and wanted web frameworks by developers in 2019.

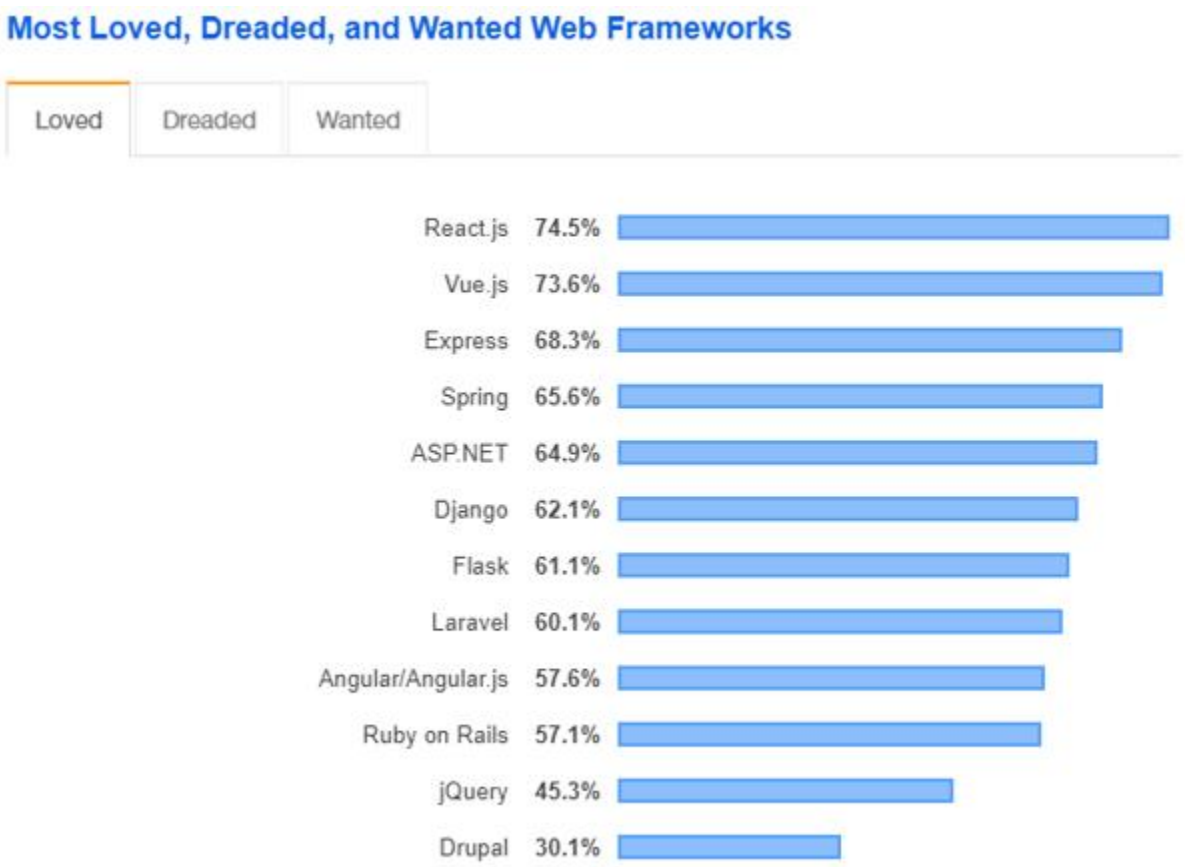

Figure 4.1: Most loved and Wanted web frameworks by developers in 2019 [36]

% of developers who are developing with the language or technology and have expressed interest in continuing to develop with it

As illustrated in the figure, Ract.js is the most loved and wanted web frameworks (74.5%) followed by Vue.js (73.6%).

To designate the component's DOM representation, React.js uses JavaScript Syntax Extension (JSX), an Extensible Markup Language (XML) like syntax extension to ECMAScript. Though JSX is JavaScript centric it looks like that of HTML or XML. Codes written with JSX is easy to read and provides a sense of writing HTML. JSX is not based on technology instead of the separation of concern. As a result, JSX includes style, behavior of components and mark-up in a single file instead of separate files. Nonetheless, it is not compulsory to use JSX in React, but using it eases error handling and development and increases performance [34].

Basic features of React.js:

- The DOM is lightweight, which improves performance
- Its Learning Curve is easy
- JSX makes it noncomplex
- High performance makes React the most loved and wanted library
- Unidirectional data flow
- Virtual DOM

As explained in the previous part of this study, the building blocks of React application are components. Like JavaScript functions, components are written with a method called React.createClass(). They enable to divide the UI of the application into separate and small reusable parts. To facilitate the communication between different components, they accept random inputs, state, and props. In most cases, props link the parent component with the child and show static immutable data. In the other hand, the state holds private and dynamic data that is controlled by the component. [39]

A tool called create react app is used in order to create and start with a new React application. Along with a preconfigured Webpack, Bable, Service Worker, and other tools create react app tool makes an app development process smooth. To install create react app from the terminal, the command called install -g create-react-app is used. Listing 4.1 below depicts a screenshot of how to install a React.js app using the terminal.

```
workneh@Workneh MINGW64 /d/annualClock/Thesis
$ create-react-app annualclock

Creating a new React app in D:\annualClock\Thesis\annualclock.

Installing packages. This might take a couple of minutes.
(node:11076) ExperimentalWarning: The fs.promises API is experimental
Installing react, react-dom, and react-scripts...

yarn add v1.6.0
info No lockfile found.
[1/4] Resolving packages...
success Saved lockfile.
success Saved 828 new dependencies.
info Direct dependencies
├─ react-dom@16.8.6
├─ react-scripts@3.0.1
└─ react@16.8.6
info All dependencies
├─ @babel/helper-builder-binary-assignment-operator-visitor@7.1.0
├─ @babel/helper-builder-react-jsx@7.3.0
Done in 471.02s.

Initialized a git repository.

Success! Created annualclock at D:\annualClock\Thesis\annualclock
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd annualclock
  yarn start

Happy hacking!
```

Listing 4.1: Installing React project using create-react-app

As shown in the listing 4.1, to create a new React app, the command create-react-app annualclock is used inside the folder annualclock. Changing the directory to annualclock and running the node package manager (npm) start or yarn start command runs the script starting the development server as shown in listing 4.2 below.

```
workneh@Workneh MINGW64 /d/annualClock/Thesis
$ cd annualclock

workneh@Workneh MINGW64 /d/annualClock/Thesis/annualclock (master)
$ yarn start
yarn run v1.6.0
$ react-scripts start
(node:45720) ExperimentalWarning: The fs.promises API is experimental
Starting the development server...

Compiled successfully!

You can now view annualclock in the browser.

  Local:            http://localhost:3000/
  On Your Network:  http://192.168.0.100:3000/

Note that the development build is not optimized.
To create a production build, use yarn build.
```

Listing 4.2: A successfully compiled Create-react-app

As illustrated in figure 4.2, the application is successfully compiled and it is running at port http://localhost:3000 and visiting the browser at that port opens the application as illustrated in Figure 4.2 below.

Figure 4.2: Running the default React project in localhost

4.2.2 NodeJS

The server-side of the application is implemented with Node.js. Unlike most popular web servers, configuring and setting up the Node server is easy and straightforward. Furthermore, unlike traditional web servers, Node is single threaded and as a result, simplifies writing web apps [40]. Node uses the JavaScript engine (Google's V8) to compile JavaScript to native machine code. Another important advantage of Node apps is that it is platform independent. It enables to run the JavaScript on the server, uncoupled from a browser and as a result, it helps to use frameworks like Express, which are written in JavaScript [12]. Nowadays, Node is becoming more popular and emerging as a dominant server-side language though there are many server-side JavaScript containers. Figure 4.3 below illustrates the list of frameworks, libraries and tools that are most loved, wanted and dreaded by developers.

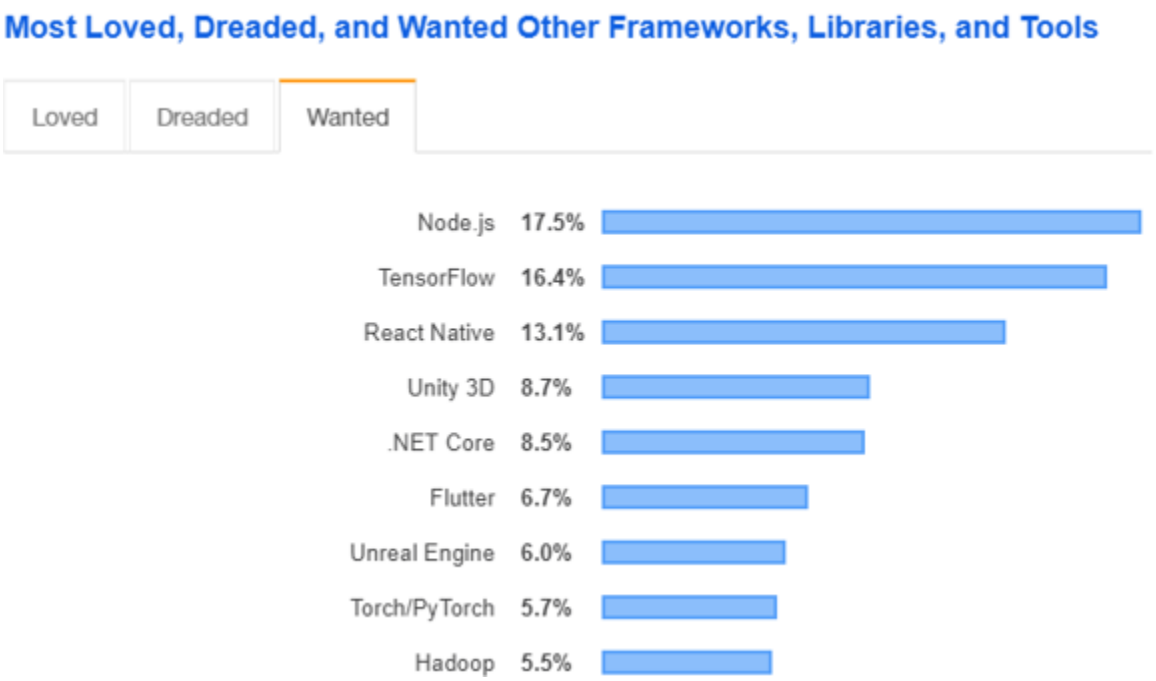

Figure 4.3: Most Wanted Other Frameworks, Libraries, and Tools [36]

% of developers using the technology or language and have shown interest to develop with it

As shown in the figure, the Node is the most wanted runtime or environment (17.5%).

5. Implementation of the Annual clock Application

The implementation of the annual clock application is described in this chapter of the thesis. The annual clock app was developed in Acer with the operating system window 8.1. Create react app and Node.js were downloaded and put together to build the application. In addition, to write the source code, visual studio code editor was used, which eases the implementation of the app because Command Line Interface (CLI) is embedded on it with built-in git support. In order to manage the node modules, NPM is used and Heroku was used to host the project.

The developed application, therefore, owns most PWAs features like: Push Notification, App Shell, Web App Manifest, Service Worker, Background Synchronization, Accessing Camera and storing data on IndexedDB.

5.1 User Interface

In this section of the study, I will explain how the front-end of the application was implemented. As stated in the above section of this study, the goal of this study was to develop an offline-first task management app and evaluate the features of PWAs.

In implementing the application, the first step was building the UI skeleton, which is the required part of any web application development. The UI was implemented using React.js and Twitter's Bootstrap. The final version of the UI was illustrated in Figure 5.1 below.

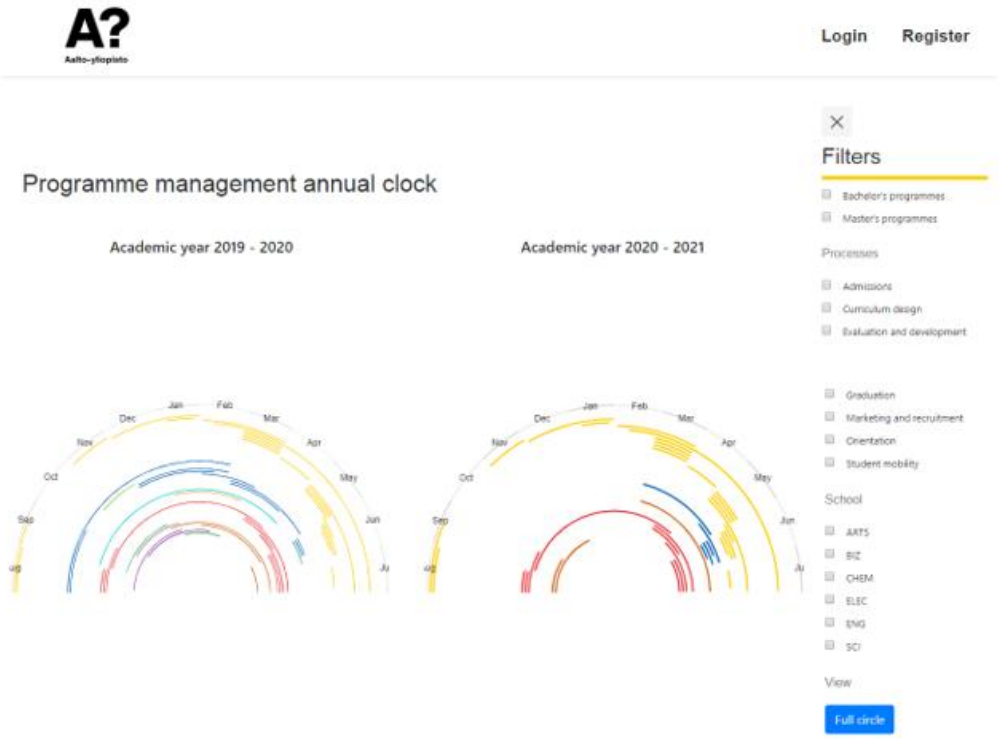

Figure 5.1: The UI of the annual clock app

As shown in the screenshot, the application has three components: application header at the top, two half-circle doughnut charts in the center and a collapsible filtering sidebar menu at the right. For the guest user, the header shows a logo, login and register links whereas for a logged in user, it has a logo, logged in username, logout link and admin link. Figure 5.2 below displays a screenshot of the registration and login pages.

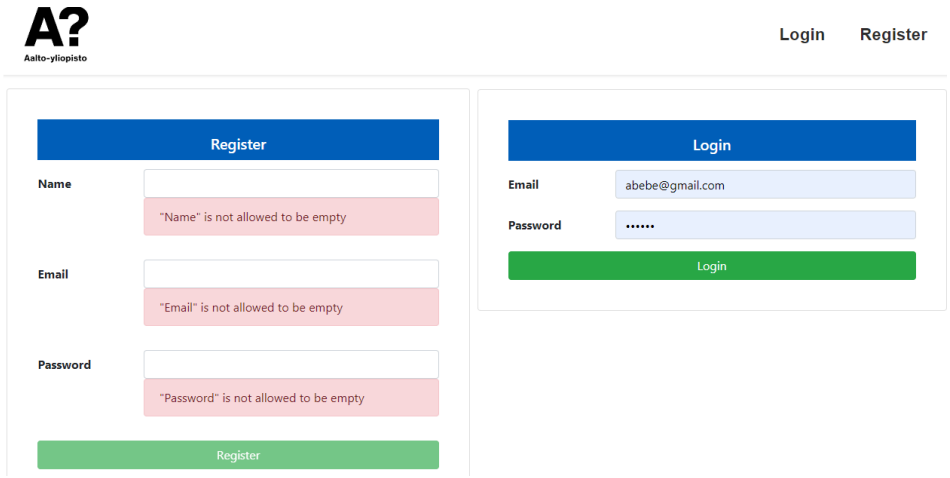

Figure 5.2: Register and Login page of the annual clock app

As shown above in the screenshot, first the admin user is required to register and login in order to perform CRUD a program. After successfully registering, the admin user login into the system and manages the admin page as illustrated in Figure 5.3 below.

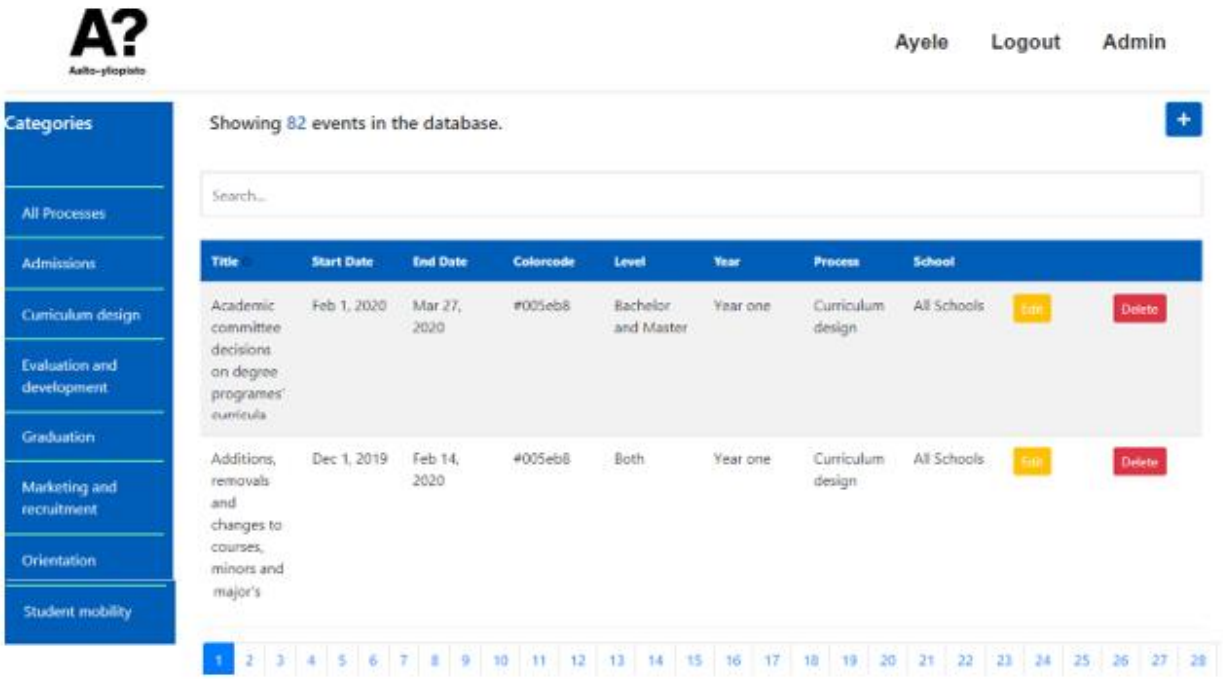

Figure 5.3: Admin page of the annual clock app

As illustrated in the above screenshot, the header of the admin page has a logo, name of the administrator, logout and admin links. The main body of the page consists of filtering sidebar menu on the left, a button with a plus icon on the top right corner, a search box, a table and a pagination link on the middle of the page. The sidebar menu on the left enables to categorize the application by processes to facilitate filtering. By default, the table lists all the processes.

The admin page has a search box on the top to manage the searching of items by title. Moreover, it is possible to sort the table by its header in ascending or descending order. Furthermore, each table body contains two buttons: edit and delete. Clicking the edit button opens the link to that specific item and clicking the delete button first shows an alert box containing a warning message and if the user accepts the warning, the item will be removed

from the application. On the bottom of the page, there are a pagination link and each page have ten items.

Finally, on the top right corner, there is a button containing a plus icon. Clicking the button opens a form which enables the admin user to create a new task. Figure 5.4 displays a screenshot of the create new event form.

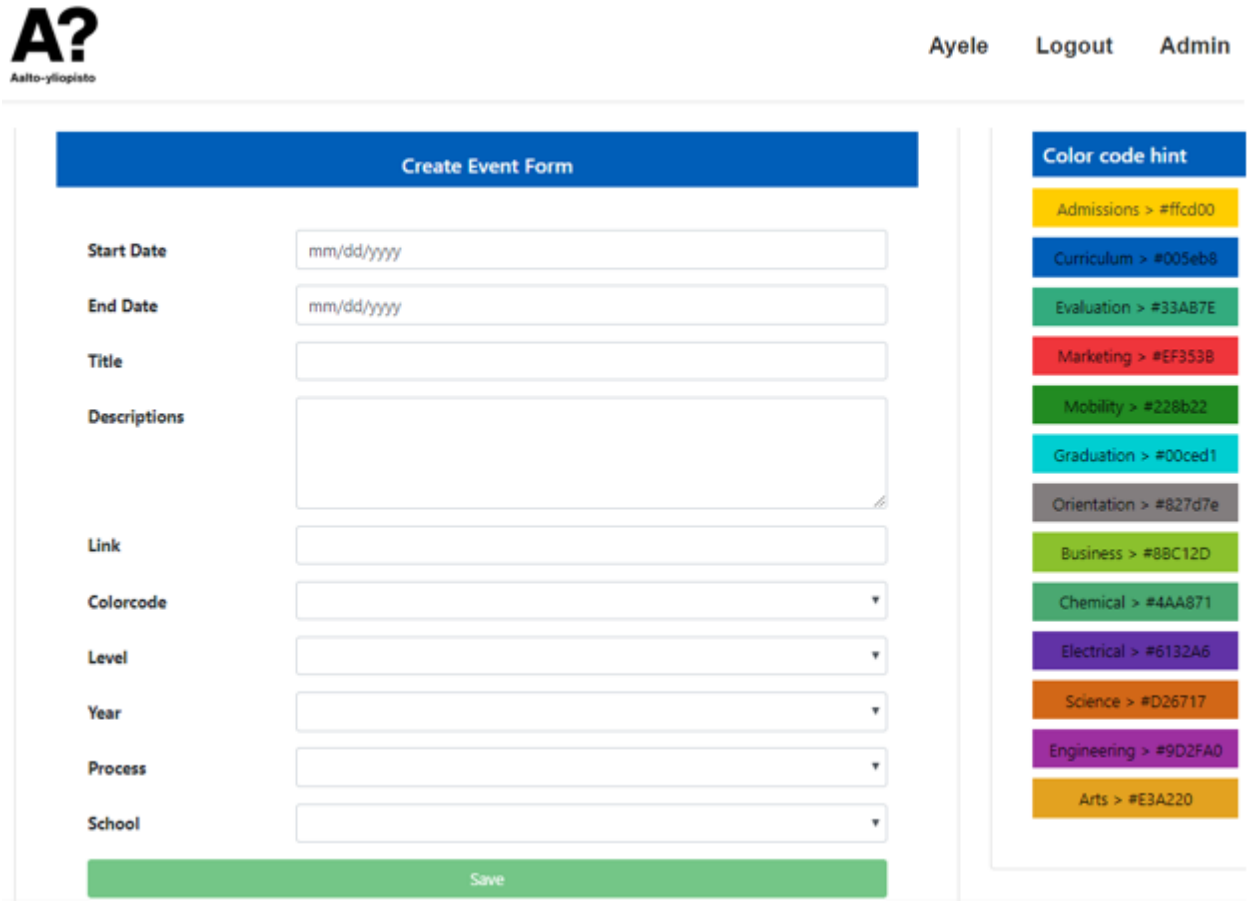

Figure 5.4: Create Event Form

As illustrated in the figure, the create event form captures important details such as start date, end date, title, descriptions, link, color code, level, year, process and school. The start and end dates are of type date, title and link are an input field but the latter accepts only valid uniform resource identifier (URI) and the descriptions field is a text area. Color code, level, year, process and school are selectable fields. About 13 types of process and school color codes are included

in the color code and the color codes are shown on the right side of the form as a hint. There are three levels: bachelor, master and both, three years, seven processes and six schools.

Concerning the doughnut chart on the home page, it has two views, half-circle view, and full-circle view. By default, it is shown as a half-circle view and its view can be changed by clicking the button on the right bottom of the sidebar menu. The filters on the right side of the home page can be used to filter the levels, processes, and schools. It is possible to select the checkbox in any combination and it shows the data in the doughnut based on that selection. For instance figure 5.5 below shows a screenshot of checking the market and recruitment checkbox in a semi-circle and full-circle view.

Figure 5.5: Filtering the marketing and recruitment process in half and full-circle view

As demonstrated in the figure, there are two doughnuts each showing two academic years. Each academic years starts on August 1 and ends on July 31. It is divided into 12 months and

the months are shown on the outer circle. Following that the duration of the events is colored by the processes or schools color. Hovering over on each section of the doughnut chart shows a tooltip showing four words from the title and clicking on each section of the chart opens a modal box containing detail information about the selected event. The screenshot in Figure 5.6 below shows the modal box.

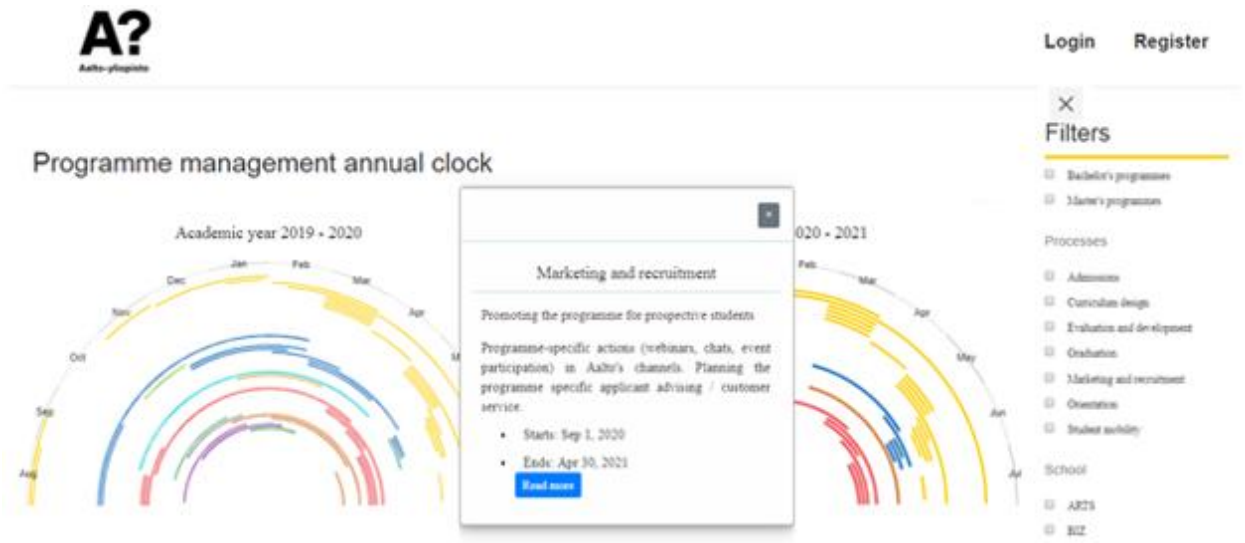

Figure 5.6: Screenshot showing contents of the modal box

As shown in the figure, the modal box contains detail information about the clicked event. It contains process name, the title of the event, description, start dates, end dates, and a link to read more about the event.

5.2 Enabling PWAs features to the annual clock app

The main advantage of PWAs is enabling the application to behave and feel like a native app. As a result, in this section of the study, I will present PWAs features like installing the app on the home screen, accessing device camera, subscribing for push notification, background synchronization, storing the data on the browser to access when the connection is lost, etc.

5.2.1 Installing the Application on the home screen

Installing the application on the home screen is one of the features of native apps. However, the user needs to know the link to the app or goes to the app store and search for the app to install a native app. Following that, the user checks the size of the app and its compatibility with the device. If the user's device has the required memory, then the user installs the application. However, in the case of PWAs, the only thing the user needs to install is the link to the web application. First, the client requests the user to install the application and if the user clicks the install button, a new browser window will be displayed showing the short name of the application on the toolbar as shown in figure 5.7 below.

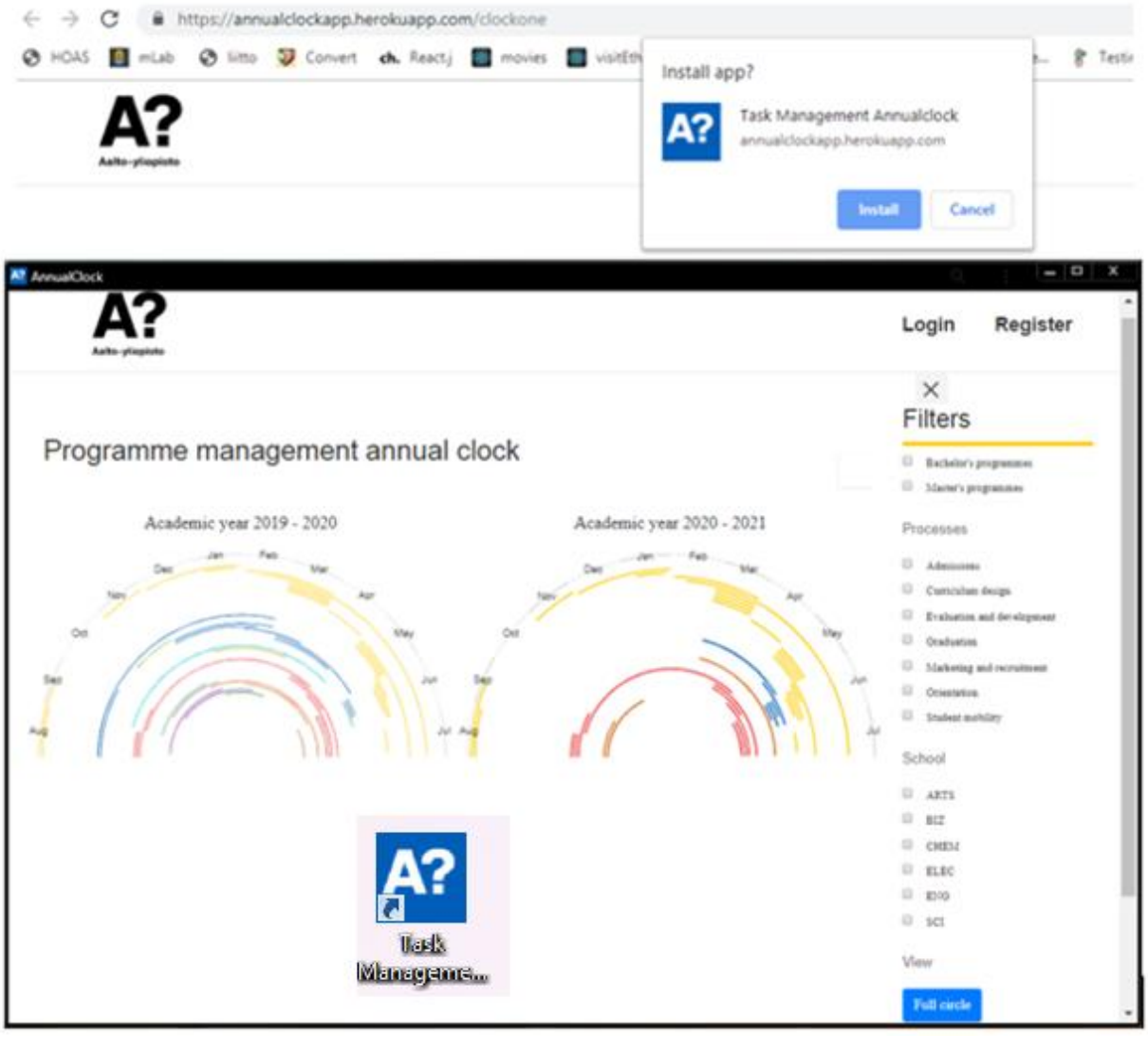

Figure 5.7: Installing the annual clock app on the home screen

As revealed in the above screenshot, an installed PWA has a standalone logo icon and a short name which are specified in the App manifest file. Moreover, a shortcut link to the app is installed in the device's home screen as illustrated in the bottom of the screenshot.

5.2.2 Accessing device camera

A link to the profile page is added and can be opened by clicking on the username. Figure 5.8 below depicts a screenshot of the profile page.

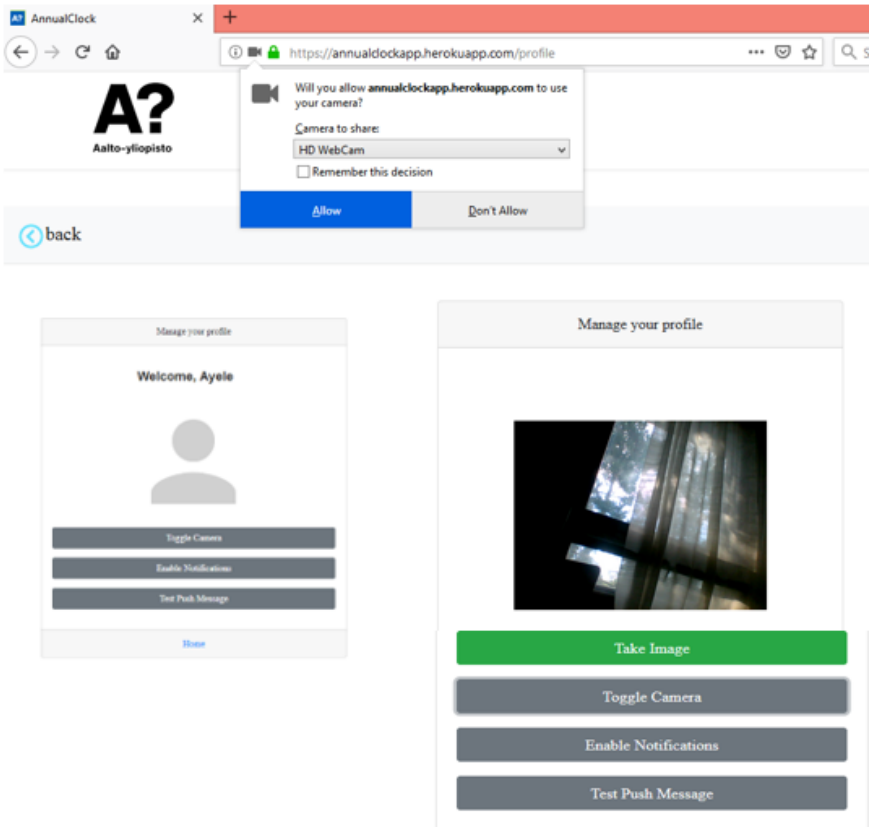

Figure 5.8: Screenshot of the profile page

As depicted in the figure, the profile page on the left contains an avatar, three buttons: toggle camera button, enable notifications button to subscribe for push notification and test push message button to test push notification. In addition, there is a back button at the top left corner of the page and a link to the home page at the bottom of the page. When the toggle camera button is clicked, the browser notifies a message to the user to enable the camera as displayed

on the top and if the user accepts the allow button, the camera will be opened and the user can capture a new image and the avatar will be replaced with the new image as shown on the right side of the screenshot.

5.2.3 Push Notification

As stated above, there is enable notification button in the profile page and when clicking the button, the browser sends a notification message and if the user accepts the message, the user will be subscribed for push notification as shown in the following figure.

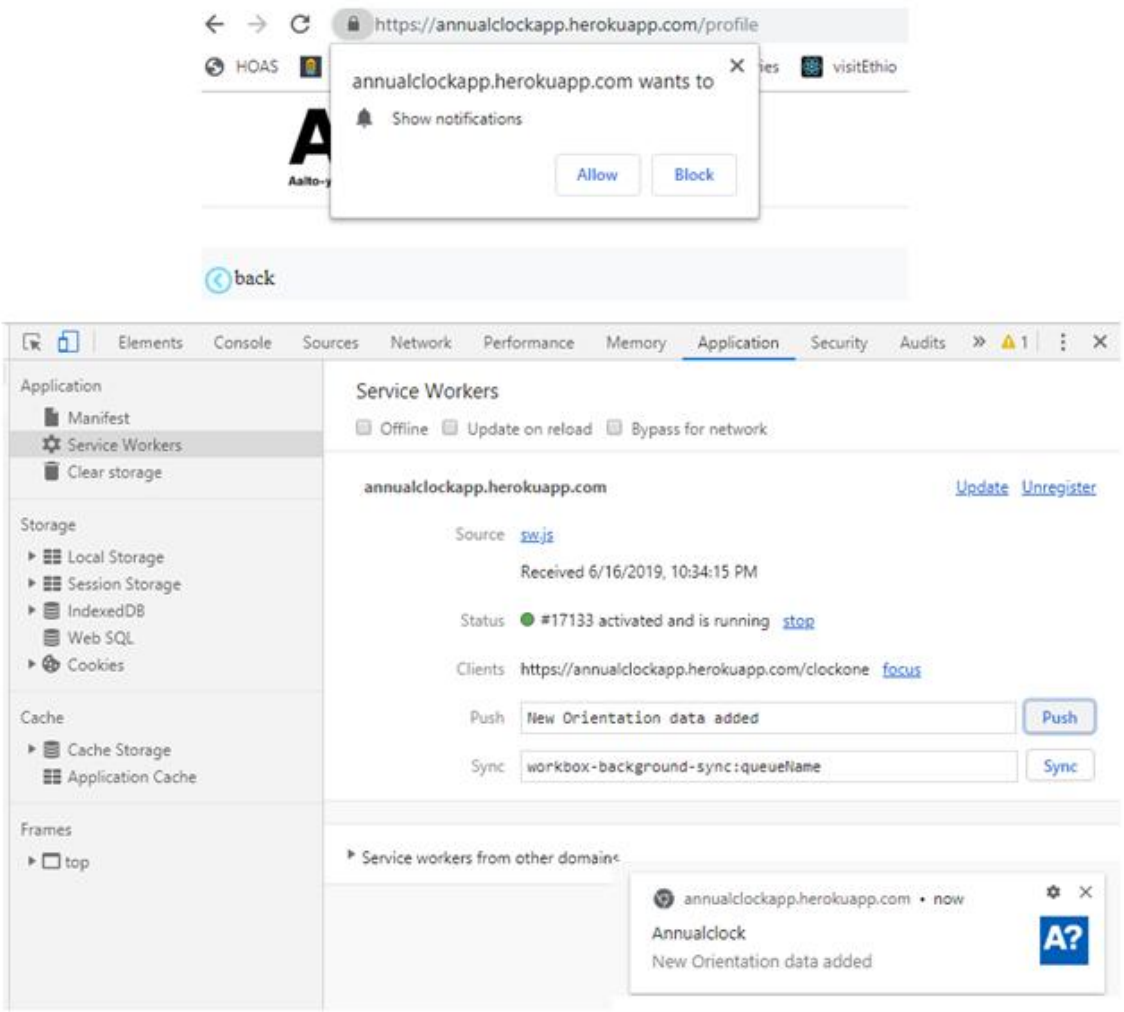

Figure 5.9: Enable push notification

As illustrated in figure 5.9, opening the chrome application tab and clicking Service Workers shows the URL (Universal Resource Locator) of the application, name of the Service Workers file, status of the Service Worker, clients, an input field to write a push and sync message. Writing a message in the push input field and clicking the push message button shows a modal box on the right bottom side containing the name of the message. Furthermore, the background sync can be made by writing the sync name and the data can be saved inside the IndexedDB.

5.2.4 Cache Storage

In order to access the application offline, all static assets should be precached inside cache storage. Figure 5.10 below depicts the contents of the cache storage of the annual clock app.

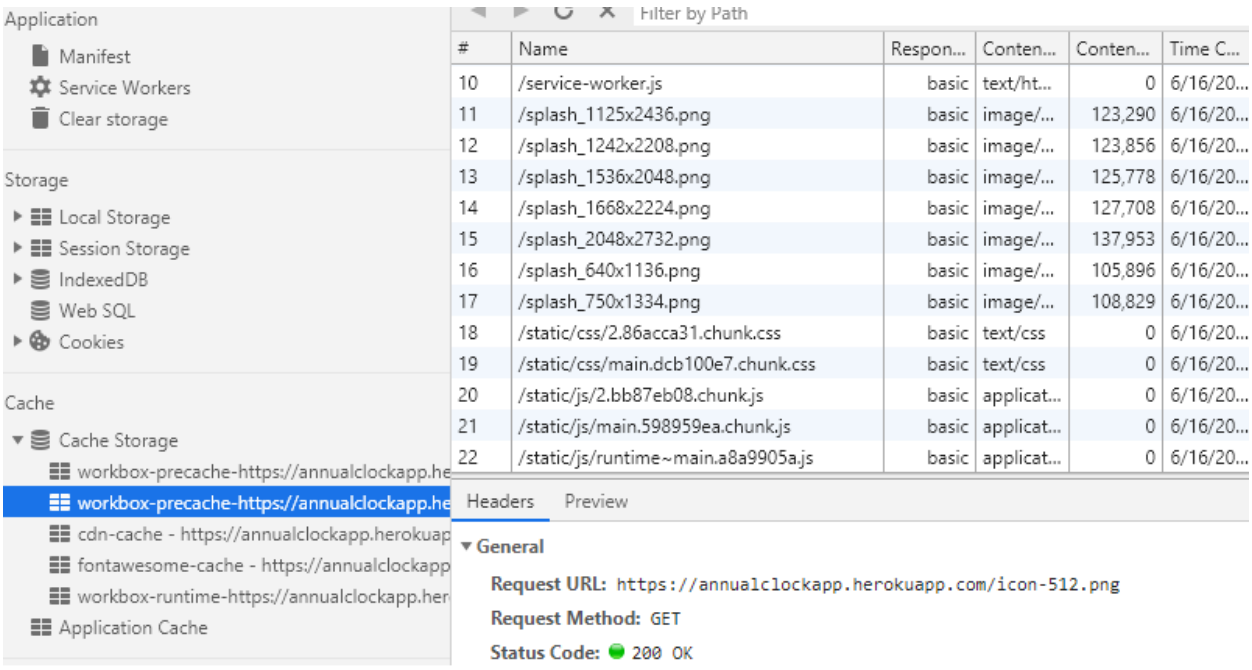

Figure 5.10: Storing all static resources in browsers Cache Storage

5.2.5 Persisting Data in IndexedDB

The cache storage saves static resources. In order to persist the dynamic data for offline use and prevent data loss, the data should be saved in the browser's storage and I choose IndexedDB for this purpose. The main purpose to choose IndexedDB over LocalStorage or

other browser storage is that it is asynchronous and does not block the UI. Furthermore, its storage is larger than that of LocalStorage and finally, it is suitable for background synchronization. Figure 5.11 below displays the contents of the IndexedDB.

Figure 5.11: Persisting dynamic data inside IndexedDB

As illustrated in the screenshot, a database called TaskDB is created inside the IndexedDB and the title is assigned as a key. When the application is online, new data added to the application will be shown in the IndexedDB. Also, it is possible to add data while the app is offline but it will be saved in the background and will synchronize when the application goes online.

5.3 Server Implementation

In this section, I will provide an overview of the server implementation. The server is implemented with JSON API specifications and the API consist of a single resource, programs. A program has ten attributes: start date, end date, title, description, link, color code, level, year, process and school.

As their name implies, start and end dates are of type dates and describe the start and end date of an event. The title is a string that describes the event and required for instance, “creation of marketing plan”. Description is also a string that describes the event in more details, but it is optional. Link is a string that accepts URI and if the user decides to read more, clicking on the link opens a new browser window. The color code is a string and used to change the background color of the doughnut chart for that specific program. About 13 color codes are included and the user selects appropriate color code for respective process or school. The level

is a string and there are three levels: bachelor, master and both. The year is of type string and used to divide the events for both academic years. Process and school are of type string and there are seven processes and six schools.

5.3.1 API Endpoints

The server has about six API endpoints for deleting, updating, creating and fetching programmes. To fetch a programme, the user can send a GET request to the appropriate API endpoints. The available API endpoints for the annual clock app are /api/programs, /api/colorcodes, /api/levels, /api/schools, /api/processes and /api/years. A 200 OK response will be responded by the server for a successful request. Figure 5.12 below shows a screenshot of /api/processes API endpoint.

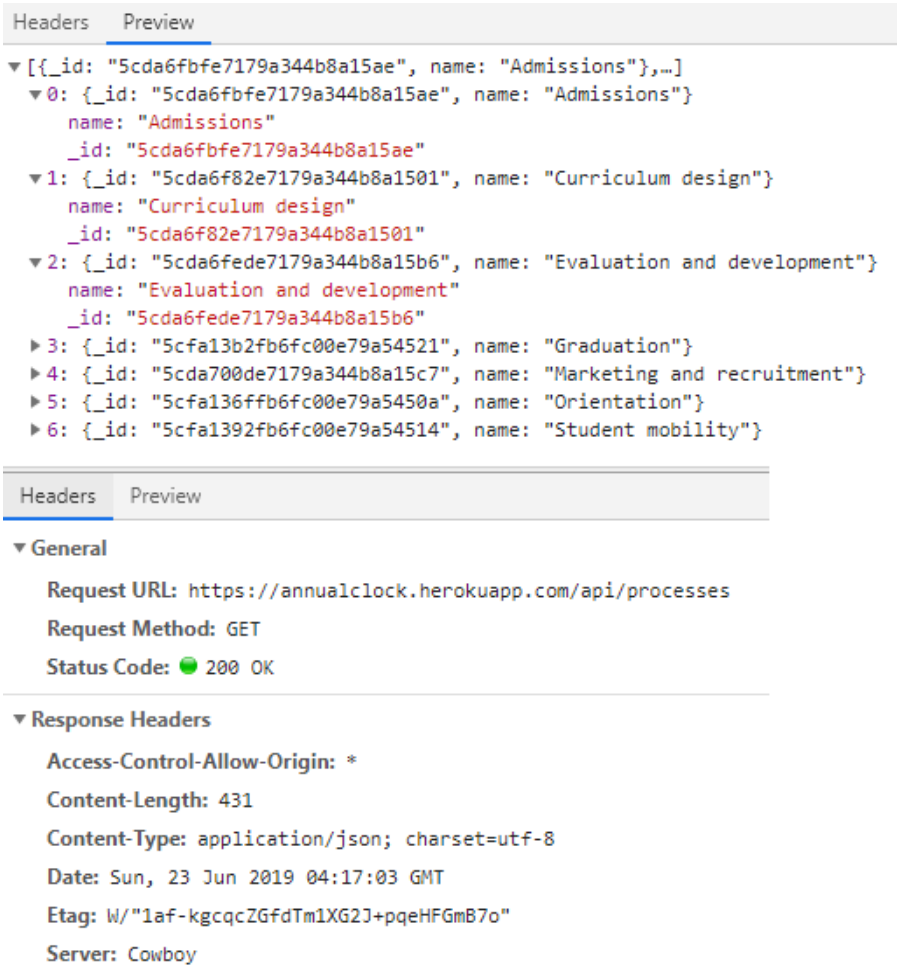

Figure 5.12: Sample API endpoints for /api/processes

To create a programme, a POST request will be sent and in the request body, the programme resource will be included. All the mandatory attributes should be included in the POST request. A PATCH request updates a programme and all or any of the programme's attributes will be updated by the request. Finally, to delete a programme, a DELETE request can be sent to the program's URL and if it is successful, a 204 No Content response can be sent by the server.

6. Evaluation of the Implemented App

In this chapter, the implemented app will be evaluated and whether the requirements set in chapter four part one will be tested. The application can be accessed offline and an audit report will be made to check whether the implemented app has PWAs features.

6.1 Accessing the Application Offline

As discussed in chapter five, the annual clock app is progressively enhanced by enabling most PWAs features. A shortcut home screen icon is installed in the mobile and desktop devices and tapping on the icon opens the application in a standalone window. Furthermore, it is possible to access the device camera and capture image and possible to subscribe for push notifications and be notified when a new data is added. In addition, the application data is saved on IndexedDB and accessing the application without any internet connection is possible.

The application was tested against two browsers: Chrome version 75 and Firefox 67.0.4. Furthermore, it was installed in the home screen of Android phone version 7.0.1 and tested as shown in figure 6.1 below.

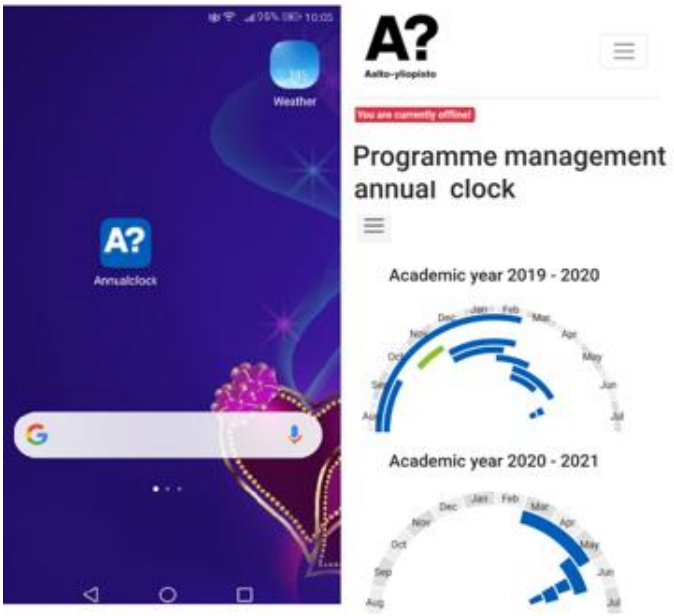

Figure 6.1: The annual clock app installed in Android phone

As demonstrated in the above figure, the annual clock app was installed in the home screen of the Android phone just like a native app and the app can be opened by tapping the icon. I tested the app offline and it works well as shown on the right-hand side of the figure. As illustrated in the figure, there is a red badge showing that the application is offline. Hence, the application works as required and all the set requirements are performed.

6.2 Testing the performance of PWAs using Lighthouse

The project was successfully configured, coded and tested and as a result, the annual clock app was successfully developed. The test was carried out with Lighthouse, an open source auditing tool built on Chrome Dev Tools that allows testing the developed web app. It can audit many very important things such as accessibility, performance, time to interactivity scores, Search Engine Optimization (SEO), PWAs, best practices, etc. The developed app was hosted in Heroku and can be found at https://annualclockapp.herokuapp.com/. The annual clock evaluation report in Lighthouse was shown in Figure 6.2 below.

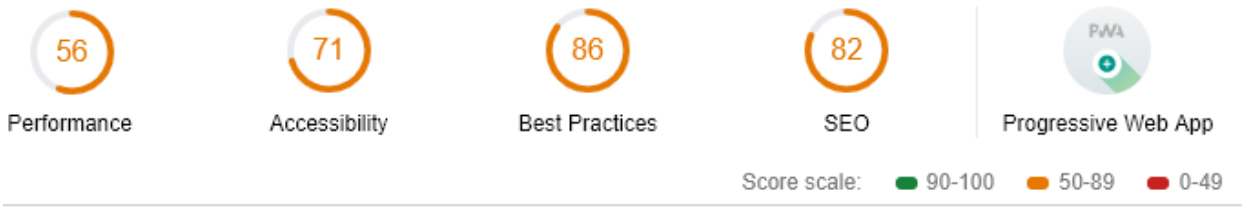

Figure 6.2. The annual clock App evaluation report in Lighthouse

As illustrated in Figure 6.2, the report showed that the developed application has 56 percent performance, 71 percent accessibility, 86 percent best practices and 82 percent SEO. Further, the application is a PWA fulfilling the planned requirement of implementing a PWA. In addition, Figure 6.3 below illustrates that the application satisfies the main aspects of PWA.

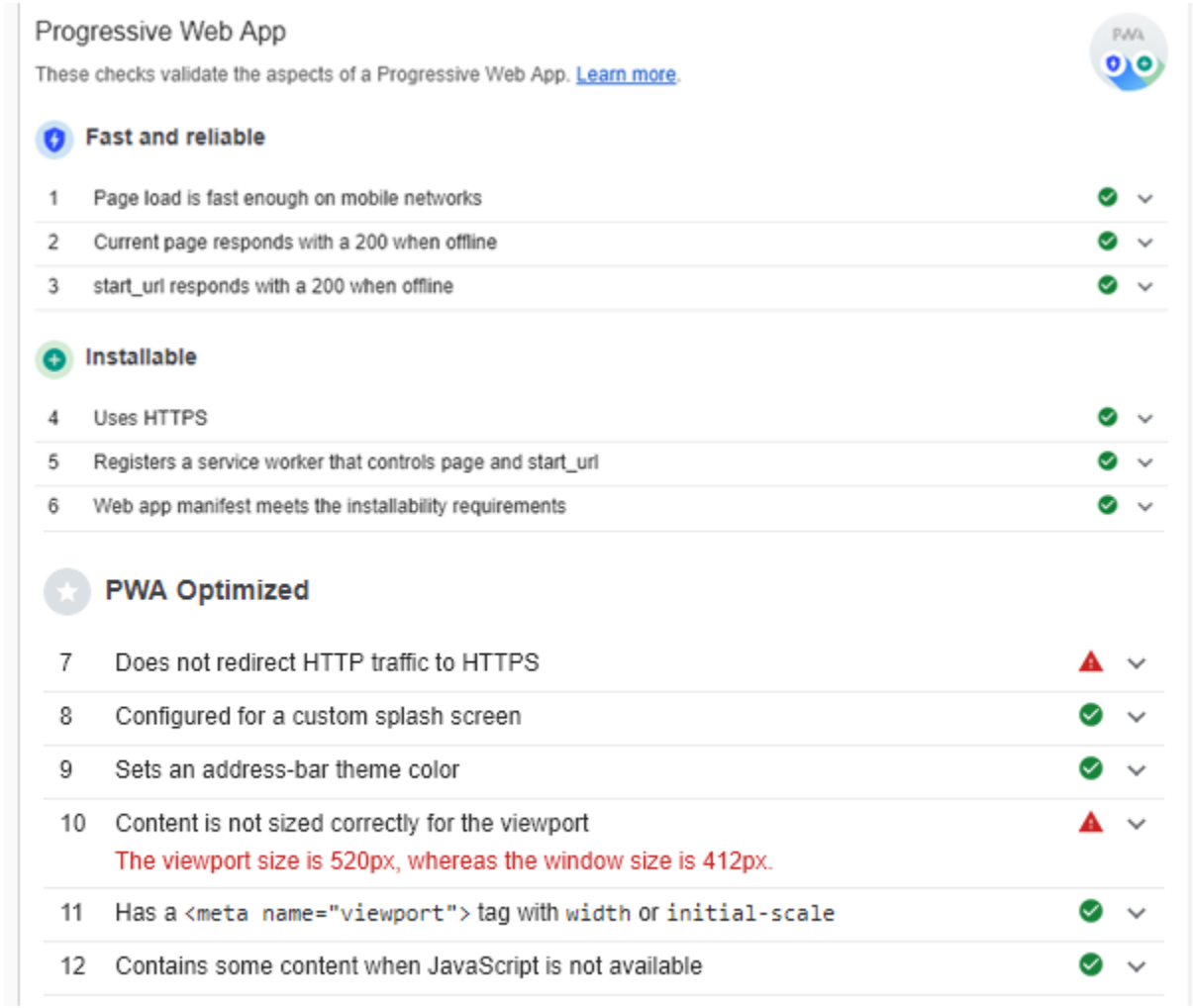

Figure 6.3. Summary of evaluation report in Lighthouse for PWA part

As displayed in the above screenshot, the application is fast and reliable as well as installable. Hence, the planned goal of implementing an annual clock app with almost all PWAs features is achieved.

7. Pros and cons of developing a PWA

In this thesis, an offline-first task management app was developed and most PWA features were enabled to the app. Hence, the planned goal of implementing an offline-first task management app and enhancing the app to feel and behave like a PWA was achieved. The application was tested and the testing result obtained in Chapter 6 illustrates that the implementation of the application met most of the conditions set in Chapter 4 Section 4.1. Nonetheless, the development of the application is not without a shortcoming and there are many areas of improvement in the implementation of the application. Hence, the purpose of this chapter is to discuss the pros and cons of developing an offline-first task management app that is enabled with PWA features.

To start with, the Application Program Interface (API) of the annual clock app consists resources like programs, processes, schools, color codes, years and levels though they are not as complex as most real-world applications. For instance, the data types for all the API endpoints do not include videos, audios, and images, unlike most real-world applications. Developing an application with these data types are challenging and time-consuming from the offline-first viewpoint. Furthermore, developing an application that uses videos, audios and images require much more storage space and challenging to store them in the browser, which has limited storage. In this application, almost all resources are cached for offline use which is unrealistic in other real-world applications that requires prioritizing what data to cache for offline use.

Another important thing, which requires due consideration in the most real-world application is software updates. In developing the annual clock app with the offline-first concept, all static resources and client-side code are cached by the client browser and there is no way for the client to know when a new version of data is added to the app. Therefore, the implementation of the application did not take into account software updates. During application development, any changes made will be seen by opening a new browser tab and refreshing the page so that the Service Worker file will be updated or by opening the application tab of the browser and forcing the Service Worker to update. However, there is no explicit mechanism for the client to update the latest version and in most case use older versions. Hence, application developers should pay due care when introducing API modifications in the server that are incompatible to the

existing ones. In order to minimize the breaking of older client versions, the API developer needs to implement API versioning or schema changes. Hence, there must be some mechanism in the client code to update the resources cached in the browser.

Concerning background synchronization, any updates from the server are fetched by the application when the page is loaded. Hence, an update from the server appears to the page when the user refreshes the page. Another issue to be considered is that the client fetches all the programs from the server though some of them are already cached in the browser. However, fetching the same data by the client considerable decrease the volume of information to be transported, minimize response time and increase server load. Hence, the proper way is to fetch the changes made in the server since the last updates which increase performance specifically in areas where internet connection is poor.

Moreover, any changes made to the application by the user while offline does not synchronize to the server until the user opens the application online. In order to notify the user whether the application is online or offline, an offline status notification message was included with a red badge on the home page. As a result, the user knows whether the app is online or offline when making changes and whether the changes are synchronized or not.

As stated in the study, the annual clock app is developed by enhancing it to feel and behave like a native app. Nonetheless, it does not mean that PWA replaces native apps instead it means a new way of developing web applications more like native apps. PWA is becoming popular these days due to its amazing features, ease of development and deployment process. Nevertheless, it is facing challenges since all features of PWAs are not yet supported by all browsers.

8. Conclusion and recommendation

Adding more features to web apps progressively to feel and work like native apps is a modern day design idea. The core concept of PWAs is enhancing web apps gradually by adding new features to existing or new web applications based on browser support. Further, implementing a PWA with the offline-first concept will enable the app to work in areas where there is no internet connection and step by step online capabilities are added to the app without avoiding the offline features. Hence, the main objective of this study was to explore PWAs features available in the browsers, which could enable developers to implement offline-first web applications.

The first two chapters introduced a background study about the web, native apps, and PWAs. Further, it reviewed the core building blocks of PWAs that are available in the browsers. Browser technologies that enable to implement offline first web applications, for instance, Service Workers, App Shell, Web App Manifest, Push Notifications, IndexedDB, Background Synchronization, and Geo-locations are researched in more detail in these two chapters. Based on the findings attained from these chapters, an offline-first task management app, the annual clock was implemented in practice.

In the literature review part of the study, I realized that some PWAs features that enable to develop offline-first web applications are not yet widely supported by all web browsers. Recent versions of Chrome supports almost all PWAs features and they are under implementation in other browsers. For instance, most recent versions of browsers support Service Workers, only Chrome version 73-75 and Chrome for Android version 71 fully support Web App Manifest, IndexedDB has globally 96.17% browser support, background synchronization is currently supported only by recent versions of Chrome and Push Notification API is supported by most known browsers excluding Internet Explorer for desktop and Safari iOS and Android WebView in mobile.

The requirement analysis made in chapter four enabled to design and implement the annual clock app, which is the main contribution of the thesis. Further, the suggestions made in pros and cons part of the study on how to improve the app are other important contributions for future

works. I did not release the source code as open source since it contains some confidential information about the university.

During implementation, I noticed that building an offline-first PWA has some challenges and there are many things need to be considered. For instance, it does not work in development mode, require to run the server for each change made on the source code, works only on secure HTTP, require to open a new browser tab to refresh the Service Worker when changes are made on the source code, etc. In addition, I presented a solution how to cache static assets and dynamic assets in the browser, how to synchronize changes in server, how to access device features like a camera in web applications and save application data in browser storage for offline use. Nonetheless, there are many areas that left untouched and need to be solved in future like saving images, audios and videos in browser storage, how to increase the size of browser storages for large data and background synchronization support by all browsers.

To sum up, the annual clock application implemented in this research study will be used as a helpful reference for the university and for other developers who is interested to develop an offline-first PWAs. Moreover, the author believe that the application serves as a foundation for investigating with clarifications to address the associated problems in the offline-first app. Finally, Aalto-university programme directors and people working with the programme can use the application to manage several processes and understand different events happening in the processes.

References

[1] Heitk¨otter, H., Majchrzak, T. A., and Kuchen, H. (2013). Cross-Platform Model-Driven Development of Mobile Applications with MD2. In Proc. 28th ACM SAC, ACM.

[2] Puder, A., Tillmann, N., and Moskal, M. (2014). Exposing native device APIs to web apps. In Proc. 1st Int. Conf. on Mobile Software Engineering and Systems. ACM.

[3] Gabor C. (2012). Tales of Creation.s [Online].

Available: http://blog.gaborcselle.com/2012/10/every-step-costs-you-20-of-users.html (Accessed 2 March 2019).

[4] I. Malavolta. (2016) Beyond Native Apps: Web Technologies to the Rescue! (Keynote). In Proc. 1st Int. Conf. on Mobile Development, Vrije Universiteit Amsterdam, Netherlands, pages 1-2. ISBN: 978-1-4503-4643-6.

[5] Andreas B., Tim A. and Tor-Morten G. (2017). Progressive Web Apps: The Possible Web-native Unifier for Mobile Development. Science and Technology Publications Lda.

[6] Russell, A., 2015. Progressive web apps: Escaping tabs without losing our soul. Infrequently Noted.

[7] Mercado, I. T., Munaiah, N., and Meneely, A. (2016). The impact of cross-platform development approaches for mobile applications from the user's perspective. In Proc. Int. Workshop on App Market Analytics, WAMA 2016. ACM.

[8] Nathan, S. (2013). Web Afternoon conference

[9] Malavolta, I., Ruberto, S., Soru, T. and Terragni, V., 2015, May. Hybrid mobile apps in the google play store: An exploratory investigation. In Proceedings of the second ACM international conference on mobile software engineering and systems (pp. 56-59). IEEE Press.

[10] Ater, T., 2017. Building progressive web apps: bringing the power of native to the browser. " O'Reilly Media, Inc.".

[11] StackOverflow: Comparison between native apps and progressive web apps. [Online]. Available: https://stackoverflow.com/questions/35504194/what-features-do-progressive-web-apps-have-vs-native-apps-and-vice-versa-on-an/39027789#39027789. (Accessed 13 March 2019).

[12] Malavolta, I., 2016, May. Web-based hybrid mobile apps: state of the practice and research opportunities. In 2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft) (pp. 241-242). IEEE.

[13] Joorabchi, M.E., Mesbah, A. and Kruchten, P., 2013, October. Real challenges in mobile app development. In 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (pp. 15-24). IEEE.

[14] Malavolta, I., Ruberto, S., Soru, T. and Terragni, V., 2015, May. Hybrid mobile apps in the google play store: An exploratory investigation. In Proceedings of the second ACM international conference on mobile software engineering and systems (pp. 56-59). IEEE Press.

[15] Kim, S.J., Wang, R.J.H. and Malthouse, E.C., 2015. The effects of adopting and using a brand's mobile application on customers' subsequent purchase behavior. Journal of Interactive Marketing, 31, pp.28-41.

[16] Google: Progressive web apps. [Online]. Available: https://developers.google.com/web/ilt/pwa/ (Accessed 16 March 2019).

[17] Yang, H.C., 2013. Bon Appétit for apps: young American consumers' acceptance of mobile applications. Journal of Computer Information Systems, 53(3), pp.85-96.

[18] Google: Progressive web app requirements for showing the "add to home screen" banner. [Online]. Available: https://developers.google.com/web/fundamentals/app-install-banners/ (Accessed 2 April 2019)

[19] Fink, G., Flatow, I. and SELA Group, 2014. Pro Single Page Application Development: Using Backbone. Js and ASP. Net. Apress.

[20] Mikowski, M. and Powell, J., 2013. Single page web applications: JavaScript end-to-end. Manning Publications Co.

[21] Champeon, S., 2003. Progressive enhancement and the future of web design. REPRINTED FROM WEBMONKEY, 1.

[22] Can I use Web App Manifest. [Online]. Available: https://caniuse.com/#search=web%20Manifest%20App. (Accessed 11 April 2019)

[23] RUSSELL, A., SONG, J., ARCHIBALD, J. and KRUISSELBRINK, M., Service Workers Nightly in M3C. 2015 [cit. 2016-12-2].

[24] Gaunt, M., 2018. Service workers: an introduction. URL: https://developers. Google. com/web/fundamentals/primers/service-workers/# cache_and_return_requests. Accessed, 20(10), p.2018.

[25] Can I use Web App Manifest. [Online] Available: https://caniuse.com/#feat=serviceworkers. (Accessed 6 April 2019)

[26] Osmani Addy. (2015). Getting started with Progressive Web Apps. [Online]. Available: https://addyosmani.com/blog/getting-started-with-progressive-web-apps/ (Accessed 20 March 2019).

[27] Mehta, N. et al. "Indexed Database API". W3C Recommendation. W3C, Jan. 8, 2015. [Online]. Available: https://www.w3.org/TR/IndexedDB/. (Accessed 16 May 2019).

[28] Can I use IndexedDB. [Online] Available: https://caniuse.com/#feat=indexeddb. (Accessed 6 April 2019)

[29] LePage Pete. Your First Progressive Web App, Google, [Online]. Available: https://developers.google.com/web/fundamentals/codelabs/your-first-pwapp/ (Accessed 20 March 2019).

[30] MDN web docs. SyncManager. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/SyncManager. (Accessed 10 May 2019).

[31] MDN web docs. Notifications browser compatibility. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/notification#Browser_compatibility. (Accessed 12 May 2019).

[32] Jake Archibald. Introducing Background Sync. [Online]. Available: https://developers.google.com/web/updates/2015/12/background-sync

(Accessed 15 May 2019).

[33] Medley, J., 2017. Web Push Notifications: Timely, Relevant, and Precise. Retrieved May 1, 2017.

[34] Fedosejev, A., 2015. React. js Essentials. Packt Publishing Ltd.

[35] Anthony, A., Nathaniel, M. and Ari, L., 2017. Fullstack React: The Complete Guide to ReactJS and Friends. Fullstack. io.

[36] Stack Overflow. Developers Survey Result 2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019. (Accessed 09 June 2019).

[37] Facebook Inc., “Introducing JSX,” [Online]. Available: https://reactjs.org/docs/introducing-jsx.html. [Accessed 22 02 2018].

[38] Aggarwal, S., 2018. Modern Web-Development using ReactJS. International Journal of Recent Research Aspects, 5, pp.133-137.

[39] ReactJS.org,’ ReactJS official'. [Online]. Available: http://www.ReactJs.org. [Accessed: 12 Jun 2018]

[40] Brown, E., 2014. Web development with node and express: leveraging the JavaScript stack. " O'Reilly Media, Inc.".